Capitolo 4

SEMAFORI

E' noto che il meccanismo dello *scambio messaggi* é il modello piú generale per rappresentare la comunicazione fra processi in quanto permette di realizzare implicitamente la sincronizzazione connessa al trasferimento dei dati fra processi. Questo risultato é ottenuto considerando i dati come *data token*, o messaggi, i quali sono costruiti associando al loro valore la cosiddetta *portante*, che rappresenta l'informazione supplementare di quando il dato é realmente disponibile per l'istruzione che lo vuole utilizzare.

Questo meccanismo, fra l'altro, permette di far comunicare processi che sono in esecuzione su processori diversi, ciascuno utilizzando una propria memoria di lavoro, in quanto non viene fatta alcuna ipotesi particolare sulla comunicazione se non la presenza di canali di comunicazione. Nel caso particolare che i processi possano condividere aree comuni di dati, come nel caso di processi in esecuzione su un'unica unitá di elaborazione e perció avendo a disposizione un'unica memoria centrale, é possibile definire un meccanismo di comunicazione semplificato in cui i processi scambiano *segnali temporali*, il cui scopo é quello di notificare l'avvenuto verificarsi di un evento.

La nozione di evento é alla base di un metodo generale di sincronizzazione fra processi che permette di risolvere qualunque problema di comunicazione fra processi purché questi condividano alcune aree di dati globali le quali sono utilizzate per il transito delle informazione e che quindi si comporatno come canali di comunicazione.

4.1 INTRODUZIONE

In molti casi i processi sono interessati a ricevere da altri processi l'informazione temporali dell'essersi verificato un certo evento. Poiché non viene scambiato alcun valore, questo si puó considerare un caso particolare di comunicazione fra processi in cui, al verificarsi di un certo *evento*, viene inviato il *messaggio vuoto*. Pertanto, se si trasferiscono soltanto *messaggi vuoti* i quali trasportano, per definizione, solo l'informazione della loro disponibilitá, questi si possono interpretare come *segnali temporali*, i quali vengono associati a cassette postali speciali piú generalmente note col nome di *semafori*, i quali sono stati introdotti da Dijkstra come meccanismo generale di sincronizzazione.

Poiché i messaggi sono vuoti e quindi indistinguibili, é sufficiente contarli cosicché la *cassetta postale* si riduce ad una singola variabile intera non negativa, che rappresenta il numero di segnali inviati ma non ancora ricevuti. Una variabile di sincronizzazione di questo tipo prende il nome di **semaforo** ed é caratterizzata, analogamente al caso delle cassete postali nell'ambito dello scambio messaggi, dalle due operazioni primitive

$$P(\text{ s }) \equiv \textbf{receive}(\text{ s, Succeded })$$

V(s) $\equiv$ **send**(s, **true**)

dove *Succeded* é una variabile booleana che indica quando l'evento, associato al semaforo, si sta verificando mentre **s** é la variabile che individua il semaforo.

Dalle definizioni precedenti segue immediatamente che l'operazione P, quando viene completata, notifica l'avvenuta ricezione del segnale prodotto attraverso la corrispondente operazione V, e che tale segnale é stato temporaneamente memorizzato dalla variabile **s**. Seguendo Habermann, un semaforo **s** risulta caratterizzato da due componenti intere:

- *sent*(s), che rappresenta il numero totale di segnali temporali ricevuti dal semaforo **s**;
- *received*(s), che rappresenta il numero totale di segnali temporali inviati dal semaforo **s**.

Poiché si é visto che lo scambio messaggi realizza la regola di sincronizzazione implicita secondo la quale i *messaggi non possono essere ricevuti prima che questi vengano spediti*, ne segue che le due variabili precedenti devono soddisfare la seguente diseguaglianza

$$0 \leq received(\ s\ ) \leq sent(\ s\ ) + \Im(\ s\ )$$

dove $\Im(\ s\ )$ sta ad indicare il numero di segnali temporali inizialmente disponibili dal semaforo s. La precedente prende il nome di **invariante della comunicazione** per il semaforo **s**.

Per quanto riguarda le regole di sincronizzazione realizzate dai semafori valgono le seguenti osservazioni. Se l'operazione P inizia nel momento in cui

$$received(s) < sent(s) + \Im(\ s\ )$$

received(s) viene incrementato di una unitá e il processo ricevente continua l'esecuzione; altrimenti, risultando

$$received(s) = sent(s) + \Im(\ s\ ),$$

questo viene ritardato fino a quando tale condizione non é soddisfatta. L'operazione V, d'altra parte, ha sempre l'effetto di incrementare sent(s) di una unitá; se uno o piú processi sono in attesa al semaforo, allora uno di questi viene selezionato per continuare l'esecuzione e received(s) viene incrementato di una unitá.

In particolare, si puó legare il valore del semaforo alle variabili intere sent(s) e received(s) secondo la relazione riportata di seguito

$$s \equiv \Im(\ s\ ) + sent(\ s\ ) - received(\ s\ )$$

dalla quale appare immediatamente che il valore del semaforo deve essere un numero intero non negativo.

4.2 DEFINIZIONE

Nella sezione precedente é stata introdotta la nozione di semaforo e si sono discusse le sue proprietá fondamentali dal punto di vista dello scambio messaggi. Siamo ora in grado di dare la seguente definizione.

Un **semaforo** *é una variabile a valori interi non negativi su cui sono definite due operazioni* P^a *e* V^b *tali che, indicato con* **s** *il semaforo,*

- *l'operazione P(* **s** *) ritarda il processo che l'ha invocata fino a quando* **s** $>$ *0 e quindi ne decrementa il valore di una unitá; scriveremo*

  P(**s** *)* $\equiv$ **await***(* **s** $>$ *0);* **s** $\leftarrow$ **s** *- 1*

- *l'operazione V(* **s** *) incrementa di una unitá il valore del semaforo; scriveremo, pertanto,*

  V(**s** *)* $\equiv$ **s** $\leftarrow$ **s** *+ 1*

Si noti che le due operazioni devono essere eseguite in modo indivisibile, cioé devono essere trattate come operazioni atomiche.

[a] prima lettera della parola olandese *passeren* (passare), in seguito sostituita con **prolagen**, contrazione di *proberen* (tentare) e *verlagen* (decrementare)

[b] prima lettera della parola olandese *vrygeven* (rilasciare), successivamente sostituita con **verhogen** (incrementare)

Poiché i semafori realizzano soltanto uno scambio di *segnali temporali* fra processi, senza che vi sia alcun transito esplicito di dati, é necessario associare ad essi un opportuno insieme di *variabili condivise* per permettere l'effettiva comunicazione fra processi. A tale scopo sono possibili due tipi di sincronizzazione noti generalmente col nome di mutua esclusione e sincronizzazione condizionata.

Nel caso della **mutua esclusione** si assicura che una sequenza di istruzioni venga eseguita come un'unica operazione indivisibile. Si consideri, ad esempio, una struttura complessa di dati manipolata da operazioni implementate come sequenze di istruzioni. Se piú processi possono eseguire piú operazioni contemporaneamente sugli stessi dati condivisi, allora puó accadere che si ottengano dei risultati non voluti.

Una sequenza di operazioni che deve essere eseguita come una operazione indivisibile si dice **sezione critica**.

Il termine mutua esclusione si riferisce all'esecuzione mutuamente esclusiva di sezioni critiche. Si noti che l'effetto di *esecuzione interposte* é visibile solamente quando due linee di computazione accedono variabili condivise. Se due *routine* non hanno variabili in comune allora la loro esecuzione non richiede di essere mutuamente esclusiva.

Un'altra situazione nella quale é necessario coordinare l'esecuzione di processi concorrenti si verifica quando una *variabile condivisa* si trova in uno stato inappropriato per permettere l'esecuzione di una particolare operazione su di essa. Qualunque processo che tenta di eseguire operazioni di questo tipo deve essere ritardato fino a quando lo stato della variabile[1] non cambia per effetto di operazioni eseguite da qualche altro processo.

Si dice **sincronizzazione condizionale** *qualunque meccanismo che ritarda l'accesso a variabili condivise fino a quando queste non si trovano in un'opportuno e ben specificato stato.*

Nelle sezioni successive verrano discussi alcuni esempi che esemplificano bene questa situazione.

4.3 IMPLEMENTAZIONE

Nei casi piú comuni i semafori sono implementati per mezzo di chiamate di sistema al *kernel*[2] il quale ha lo scopo di realizzare la multiprogrammazione su un singolo processore. In ogni istante, ciascun processo é *pronto* per andare in esecuzione sul processore oppure é *bloccato* in attesa che venga completata un'operazione *P* come appare evidente nella definizione data nella sezione precedente.

[1] cioé l'insieme dei valori che caratterizzano quella variabile

[2] noto anche col termine di *supervisore* o *nucleo*

La maggior parte delle implementazioni dei semafori assumono di soddisfare la condizione di **equitá**[3] secondo la quale *nessun processo ritardato in seguito all'esecuzione di un'operazione* P *rimane in attesa per sempre anche se vengono eseguite operazioni* V *infinitamente spesso.*

Chiaramente la condizione di equitá ha significato quando un certo numero di processi vengono ritardati durante il tentativo di completare un'operazione sul medesimo semaforo **s** poiché, in questo caso, l'implementazione é costretta a scegliere quale processo sará autorizzato a proseguire.

Volendo ora implementare i semafori, potremmo supporre di avere a disposizione due porte, indicate rispettivamente con PService e VService, e che corrispondono alle due operazioni P e V. Al semaforo viene associata una struttura dati comprendente un valore intero[4] e una coda di processi[5] in modo tale che le operazioni siano immediatamente realizzabili dal Kernel nel modo che appare nel seguito.

Come avviene normalmente nell'ambito scambio messaggi le procedure P e V sono realizzate in termini delle primitive send e receive con i canali di comunicazione che si deducono dal codice riportato di seguito.

```
type Semaphore = ↑ Descriptor;

        Descriptor = record
                          Val : integer;
                          Qu : Queue
                     end;

procedure P( var s : Semaphore );
var     PId : Process;
        Done : boolean;
begin GetPid( PId );
    send( PService, s, Pid );
    receive( Kernel, Done )
end;

procedure V( var s : Semaphore );
begin
    send( VService, s )
end;

procedure Kernel;
var PId : Process;
begin
    InitS;
    do true; receive( PService, s, Pid ) → with s↑ do
        if ( Val > 0 ) then
            begin
                Val := Val - 1;
                send( PId, true )
            end else
```

[3]nota col termine inglese di *fairness*

[4]il valore del semaforo

[5]in attesa che il semaforo assuma un valore strettamente positivo

```
            Insert( Qu, PId );
    [] true; receive( VService, s ) → with s↑ do
        if IsEmpty( Qu ) then
                Val := Val + 1 else
            begin
                Remove( Qu, Pid );
                send( PId, true )
            end
    enddo;
end;
```

Come é facile intuire l'implementazione dipende dal modo con cui sono gestite le code di processi associate ai semafori.

4.4 PROVE DI CORRETTEZZA

I semafori sono uno strumento molto generale per risolvere problemi di sincronizzazione. Sfortunatamente tali soluzioni tendono a mostrare una notevole complessitá anche in presenza di problemi non particolarmente articolati come, ad esempio, il problema dei lettori/scrittori. Questo dipende dal fatto che il numero dei semafori tende a proliferare quando i processi coinvolti necessitano di scambiare informazioni in diversi stati della loro computazione.

Nel seguito vengono definiti alcuni problemi classici, presentandone la soluzione accompagnata da alcune argomentazioni *semiformali* per convincerci della correttezza della soluzione. Queste si basano sull'utilizzo dell'invariante di semaforo e su alcune relazioni che si evincono dalla forma del codice della soluzione.

4.4.1 Il Problema della Mutua Eslusione

Per realizzare la soluzione al problema della mutua esclusione ogni sezione critica é preceduta da un'operazione P ed é seguita da un'operazione V sullo stesso semaforo. Tutte le sezioni critiche della medesima classe[6] usano lo stesso semaforo, inizializzato ad uno. Poiché tali semafori prendono soltanto i valori zero e uno sono spesso chiamati **semafori binari**.

Vogliamo verificare ora che la soluzione del problema della mutua esclusione soddisfa le condizioni appena citate, che possono schematizzarsi nel codice seguente

```
........
P( Mutex );
< Sezione Critica >;
V( Mutex );
........
```

dove, all'invariante del semaforo *Mutex* inizializzato ad uno,

$$(a) \quad received(Mutex) \quad \leq \quad sent(Mutex) + 1$$

[6] cioé, quelle caratterizzate da sezioni di codice che si riferiscono allo stesso insieme di variabili condivise

si deve aggiungere la seguente diseguaglianza

$$(b) \quad received(Mutex) \quad \geq sent(Mutex);$$

che si evince dalla forma del protocollo che implementa la soluzione al problema in esame. In questo modo, in virtú della catena di diseguaglianze (a) e (b) si ottiene

$$sent(Mutex) \leq received(Mutex) \leq sent(Mutex) + 1$$

e, quindi, sottraendo membro a membro,

$$0 \leq received(Mutex) - sent(Mutex) \leq 1$$

che scriveremo, anche, in forma compatta

$$0 \leq inside(Mutex) \leq 1$$

avendo definito con *inside(s)* ≡ *received(s)* - *sent(s)* il numero di processi che stanno eseguendo la propria sezione critica. Si vede, allora, come il valore unitario di inizializzazione del semaforo *Mutex* sia basilare nel stabilire la correttezza del protocollo d'accesso alla regioni critica.

4.4.2 Il Problema del Produttore/Consumatore

Nell'implementazione della *sincronizzazione condizionale* si devono impiegare sia variabili condivise per rappresentare condizioni, sia di semafori associati ad esse per realizzare la sincronizzazione. Appena un processo ha reso la condizione vera, lo segnala eseguendo l'operazione V; con l'operazione P un processo ritarda la sua esecuzione in attesa che la condizione diventi vera.

Un semaforo che puó prendere un qualunque valore non negativo si dice anche *semaforo generale* o *semaforo contatore*. Semafori di questo tipo vengono spesso impiegati nella realizzazione di *sincronizzazioni condizionali* quando si deve gestire l'allocazione di risorse. In questo caso il semaforo prende un valore iniziale pari al numero di unitá della risorsa; l'operazione P serve a ritardare i processi fino a quando non sia disponibile un'istanza libera della risorsa mentre l'operazione V ha come effetto quello di segnalare la disponibilitá di una risorsa. Ovviamente, se é disponibile un'unica istanza di risorsa, allora é sufficiente l'impiego di semafori binari.

Il semplice esempio che segue illustra chiaramente quest'uso dei semafori. Si consideri, allora, una coppia di processi che possono scambiarsi risultati parziali mediante un *buffer* di comunicazione, organizzato come una coda circolare di lunghezza *N*. Si supponga, inoltre, di poter accedere alle locazioni del buffer mediante due operazioni primitive Put e Get, implementate secondo il codice riportato di seguuito.

```
const BufferSize = N;
type  Index = 0 .. BufferSize-1;
      Data = array[ Index ] of Item;
      Buffer = record
                 Dx : Data;
                 ld : Index;
                 lf : Index;
               end;
```

```
procedure Put( var Bu : Buffer; Msg : Item );
begin
   with Bu do
      begin
         Dx[ ld ] := Msg;
         ld := ( ld + 1 ) mod BufferSize
      end
end;

procedure Get( var Bu : Buffer; var Msg : Item );
begin
   with Bu do
      begin
         Msg := Dx[ lf ];
         lf := ( lf + 1 ) mod BufferSize
      end
end;
```

Poiché le sezioni di codice che controllano l'accesso al Buffer sono critiche per i due processi, é necessario prefissare le operazioni di inserzione e prelievo con opportuni protocolli. Si supponga, a questo proposito, di impiegare il seguente schema di implementazione

```
var   NotFull : semaphore;
      NotEmpty : semaphore;

procedure Deposit( var Bu : Buffer; Msg : Item );
begin
   P( NotFull );
   Put( Bu, Msg );
   V( NotEmpty )
end;

procedure Fetch( var Bu : Buffer; var Msg : Item );
begin
   P( NotEmpty );
   Get( Bu, Msg );
   V( NotFull )
end;
```

che utilizza una coppia di semafori in modo tale che

- il semaforo *NotFull* ritarda l'esecuzione del produttore fino a quando il buffer ha almeno una locazione disponibile, ossia, non è pieno,
- il semaforo *NotEmpty* ritarda l'esecuzione del consumatore fino a quando non sia disponibile almeno un dato da prelevare, ossia, il buffer non é vuoto.

Volendo verificare la correttezza della soluzione proposta, si deve esplicitare per ciascun semaforo il relativo invariante e stabilire le restrizioni che devono essere verificate, in conseguenza della forma delle operazioni Deposit e Fetch in riferimento ai segnali temporali scambiati fra i due semafori *NotFull* e *NotEmpty*.

Si ricorda, a questo proposito, che indicato con s il nome di un semaforo, deve essere soddisfatta la relazione di invarianza del semaforo,

$$received(s) \leq sent(s) + \Im(s)$$

dove

- *sent*(s) rappresenta il numero di segnali temporali inviati al semaforo s dai processi che hanno completato una operazione **V**;
- *received*(s) rappresenta il numero di segnali temporali, provenienti dal semaforo s, che sono ricevuti dai processi che hanno completato una operazione **P**;
- $\Im$(s) rappresenta il numero di segnali temporali inizialmente disponibili dal semaforo s.

Pertanto, nel caso in esame, devono essere sempre verificati i due invarianti di semaforo

$$\begin{array}{lllll} (a) & received(NotFull) & \leq & sent(NotFull) + N; \\ (b) & received(NotEmpty) & \leq & sent(NotEmpty); \end{array}$$

mentre, avendo indicato con d il numero di operazioni di inserzione di dati nel buffer e con f il numero di operazioni di prelievo di dati, la struttura delle procedure Fetch e Deposit porta a scrivere le diseguaglianze

$$\begin{array}{lllllll} (c) & received(NotFull) & \geq & d & \geq & sent(NotEmpty); \\ (d) & received(NotEmpty) & \geq & f & \geq & sent(NotFull); \end{array}$$

In questo modo, in virtú della catena di diseguaglianze (d), (b), (c), si ottiene

$$f \leq received(NotEmpty) \leq sent(NotEmpty) \leq d$$

mentre, utilizzando le diseguaglianze (c), (a), (f), si puó scrivere

$$d \leq received(NotFull) \leq sent(NotFull) + N \leq f + N$$

che possono essere combinate nella diseguaglianza seguente

$$f \leq d \leq f + N$$

da cui é immediato ricavare, per sottrazione di egual quantitá da ambo i membri,

$$0 \leq d - f \leq N$$

la quale sta a significare che lo spazio disponibile, *pari alla differenza fra il numero di messaggi inseriti e quelli estratti*, non puó eccedere le dimensioni del buffer.

Infine, si puó constatare che le operazioni di inserzione ed estrazione non interferiscono mai fra loro. A questo proposito é sufficiente verificare che i valori d e f definiti al punto precedente, non si riferiscono mai alla stessa locazione del buffer, ossia, la condizione $(d \equiv f)_{modN}$ non é mai verificata.

La forma delle operazioni *Deposit* e *Fetch* suggerisce che il numero d di inserzioni e il numero f di estrazioni sono uguali al numero di operazioni **V** completate, rispettivamente, sui semafori *NotEmpty* e *NotFull*. Pertanto, possiamo scrivere le seguenti relazioni,

$$\begin{array}{lll} (e) \quad d = & sent(NotEmpty) & = received(NotFull) - 1; \\ (f) \quad f = & sent(NotFull) & = received(NotEmpty) - 1; \end{array}$$

dalle quali é immediato ricavare

$$\begin{array}{llllll} d = & received(NotFull) - 1 & \leq & sent(NotFull) + N - 1 & \leq & f + N - 1 \\ f = & received(NotEmpty) - 1 & \leq & sent(NotEmpty) - 1 & \leq & d - 1 \end{array}$$

e, quindi, combinando le precedenti,

$$1 \leq d - f \leq N - 1$$

la quale afferma che lo spazio compreso fra la prima locazione libera e il primo *item* disponibile é sempre non nullo e inferiore alla dimensione del buffer.

4.4.3 Mutua Esclusione Selettiva

Si possono utilizzare i semafori anche per risolvere il problema della *mutua esclusione selettiva*. In questo caso le variabili condivise vengono partizionate in sottoinsiemi disgiunti, associando un semaforo a ciascun sottoinsieme con lo stesso protocollo visto nel caso della semplice mutua esclusione. Le sezioni critiche che si riferiscono a variabili appartenenti allo stesso sottoinsieme sono esesguite in regime di mutua esclusione mentre le sezioni critiche che si riferiscono a variabili di sottoinsiemi differenti, posssono essere eseguite concorrentemente.

Bisogna tener presente, comunque, che se due o piú processi richiedono il simultaneo accesso a variabili di due o piú sottoinsiemi, allora é possibile che si verifichino condizioni di stallo qualora non si prendano particolari precauzioni sul protocollo d'accesso all'insieme delle variabili condivise nella sua globalitá.

Si supponga, ad esempio, che due processi P_1 e P_2 richiedano un accesso simultaneo agli insiemi di variabili condivise A e B. Allora, P_1 e P_2 potrebbero trovarsi in una situazione di stallo se a P_1e P_2 fosse consentito l'accesso agli insiemi A e B, rispettivamente, ed inoltre entrambi ricevessere l'assenso ad accedere all'altro insieme di variabili.

4.5 NUCLEO DI UN SISTEMA DI CALCOLO

L'implementazione dei semafori mediante scambio messaggi su porte mette chiaramente in luce il fatto che il nucleo implementa i semafori in modo tale che ciascun processo, nel corso della sua esistenza, si puó trovare in uno *stato* di pronto, ed andare in esecuzione sul processore, oppure, in uno *stato* bloccato in attesa di completare un'operazione P.

Le funzionalitá del nucleo sono realizzate, generalmente, da due componenti specifiche del nucleo, che prendono il nome di *scheduler* e *dispatcher*. Il primo gestisce la *lista dei processi pronti*, organizzata come una coda di descrittori di processo, e distribuisce[7] i flussi di controllo di ciascuno di essi sull'unica unitá centrale di elaborazione, facendo eseguire ogni processo sequenziale per un periodo di tempo fissato. Il secondo consente la riattivazione e la sospensione effettiva dei processi mediante salvataggio e ripristino dei **contesti**, definiti come l'immagine in memoria di tutti i registri utilizzati dalla CPU e, fra questi, il program counter.

[7]generalmente noto col termine inglese *multiplexing*

4.5.1 Gestione dei Processi

I descrittori dei processi che sono bloccati su un certo semaforo vengono spostati dalla coda dei processi pronti a quella associata al semaforo in modo che non possano andare in esecuzione. Le operazioni P e V sono normalmente realizzate come `trap` a routine del nucleo. In corrispondenza di un'operazione **P**, se il semaforo ha un valore positivo, il suo valore viene decrementato; altrimenti il descrittore del processo in esecuzione viene spostato nella coda associata al semaforo. L'operazione **P**, invece, provoca lo spostamento di un descrittore dalla coda del semaforo, se non é vuota, a quella dei processi pronti; altrimenti, il valore del semaforo viene incrementato di un'unitá.

Il codice riportato nel seguito schematizza il funzionamento del nucleo relativamente alle operazioni di semaforo, le quali sono implementate in termini delle operazioni fornite dallo *scheduler* e dal *dispatcher*.

```
type Semaphore = ↑ Descriptor;

     Descriptor = record
                        Val : integer;
                        Qu : Queue
                  end;

procedure P( var s : Semaphore );
var  Curr : Process;
     Next : Process;
begin Lock( CPU );
   with s↑ do
      if ( Val = 0 ) then
         begin
            Suspend( Curr );
            SaveContext( Curr );
            Insert( Qu, Curr );
            Awake( Next );
            LoadContext( Curr );
         end else
         Val := Val - 1
   Unlock( CPU );
end;

procedure V( var s : Semaphore );
var  Pid : Process;
begin
   Lock( CPU );
   with s↑ do
      if not IsEmpty( Qu ) then
         begin
            Remove( Qu, Pid );
            Activate( Pid );
         end else
         Val := Val + 1
```

```
        Unlock( CPU );
    end;
```

In esso compaiono un certo numero di procedure predefinite che rappresentano le azioni dello scheduler, del dispatcher e del gestore delle code associate ai semafori, secondo quanto precisato nel seguito.

- **scheduler**, responsabile della gestione della coda dei processi pronti attraverso le seguenti procedure
  - *Suspend*, sospende il processo correntemente in esecuzione, rimuovendolo dalla coda dei pronti;
  - *Activate*, introduce il processo indicato nella coda dei pronti;
  - *Awake*, risveglia un processo pronto mandandolo in esecuzione
- **dispatcher**, responsabile della gestione dei contesti con le seguenti procedure
  - *SaveContext*, salva i registri della CPU, relativamente al processo correntemente in esecuzione, sotto forma di contesto per essere riattivato in un istante successivo;
  - *LoadContext*, carica i registri della CPU secondo il contenuto del contesto indicato, ripristinando l'esecuzione di quel processo
- **gestore delle code dei semafori**, con le procedure
  - *IsEmpty*, verifica se la coda in questione é vuota;
  - *Remove*, rimuove dalla coda in esame un processo secondo la politica di gestione della coda (ad esempio, FIFO);
  - *Insert*, inserisce nella coda in questione il processo indicato.

Comunque, é interessante notare come le operazioni sui semafori debbano essere realizzate in condizioni di interruzione disabilitate, per garantirne l'atomicitá. Questo perché l'accesso alla struttura globale dei descrittori di processo deve avvenire in condizioni di mutua esclusione onde evitare che le informazioni ivi contenute non vengano modificate in modo incompleto, portando l'intero insieme di variabili condivise, che rappresenta lo stato di esecuzione di tutti i processi sul processore, in uno stato inconsistente. L'esempio che segue mette chiaramente in luce tali problematiche.

4.5.2 Gestione delle Interruzioni

Si consideri la struttura di un generico sistema di calcolo costituito da una *CPU* ed un certo numero di *device controller* , uno per ciascun dispositivo connesso con la CPU. Si supponga, in particolare, di considerare il *controller* di un disco (*SDController*) la cui forma generale, come é noto, é la seguente

```
procedure SDController;
{ dichiarazione di variabili locali }
  begin
    { inizializzazione }
    repeat
      receive( SDDriver, SDBusy );
      Examine( Op, Cyl, Sec, Tr );
      Execute( Op, Cyl, Sec, Tr );
```

```
            send( InterruptVector, SD )
        until false
    end;
```

dove

- *SDDriver* é il driver del dispositivo, identificato dal nome simbolico DK0,
- *SDBusy* é la linea di connessione fra controller e CPU che permette a quest'ultima di attivare il controller per un nuovo ciclo di operazioni,
- *Examine* permette di esaminare nel registro-istruzioni del controller il tipo e gli operandi dell'operazione,
- *Execute* realizza l'effettiva esecuzione dell'istruzione, i cui operandi identificano univocamente numero di cilindro, di settore e di traccia del blocco da trasferire,
- *InterruptVector* rappresenta il vettore delle interruzioni, la cui dimensione eguaglia il numero di dispositivi noti alla CPU.

Si supponga di realizzare le operazioni **ReadBlock** e **WriteBlock** che leggono, rispettivamente, scrivono un blocco sul disco SD, come *chiamate di procedura remota* secondo lo schema seguente

```
procedure ReadBlock( Blk : Block; var Base : Address );
var Proc : Process;
begin
    GetPId( Proc );
    send( SD, Proc, ReadOp, Blk );
    receive( SDDriver, Base )
end;
procedure WriteBlock( Blk : Block; var Base : Address );
var Done : boolean;
      Proc : Process;
begin
    GetPId( Proc );
    send( SD, Proc, WriteOp, Blk, Base );
    receive( SDDriver, Done )
end;
```

dove

- *SD*, rappresentando il nome logico del disco, é la *porta* cui i processi inviano le richieste di accesso al disco come operazioni di lettura (ReadOp) e/o scrittura (WriteOp);
- *Blk* identifica l'indirizzo logico del blocco su disco;
- *Base* é l'indirizzo del blocco in memoria centrale, immagine del blocco residente su disco.

Il driver del disco SD é realizzato come un *server* che riceve richieste dall'unitá logica SD le quali, dopo essere state processate, vengono passate all'unitá di controllo SDController e che é pilotato, a sua volta, dalla routine di risposta alle interruzioni. Nella soluzione, presentata di seguito, si é definito, dapprima, il server *InterruptHandler* delle routine di risposta alle interruzioni che attende, sulla porta *InterruptVector*, il nome del dispositivo che lancia l'interruzione. Per comoditá si sono supposti connessi i seguenti dispositivi: *Cons* (console), *Tty* (porta seriale), *SD* (unitá disco) e *ST* (unitá nastro).

```
const MaxProc = MAXPROCESS;
      MaxCyl = MAXCYL;
      MaxSector = MAXSECTOR;
      MaxTrack = MAXTRACK;
      MaxBlock = MAXBLOCK;
      MemSize = MEMSIZE;

type Process = 0 .. MaxProc;
      Cylinder = 0 .. MaxCyl;
      Sector = 0 .. MaxSector;
      Track = 0 .. MaxTrack;
      Block = 0 .. MaxBlock;
      Address = 0 .. MemSize;
      DiskOperation = ( ReadOp, WriteOp );

procedure SDController;
var Op : DiskOperation;
      Cyl : Cylinder;
      Sec : Sector;
      Tr : Trace;
      Loc : Address;
begin
   InitSD;
   repeat
      receive( SDDriver, SDBusy );
      Examine( Op, Cyl, Sec, Tr );
      Execute( Op, Cyl, Sec, Tr );
      send( InterruptVector, SD )
   until false
end;

procedure InterruptHandler;
begin
   AttachDevices;
   repeat
      receive( InterruptVector, DName );
      case DName of
         Cons : send( ConsDriver, true );
         Tty : send( TtyDriver, true );
         SD : send( SDDriver, true );
         ST : send( STDriver, true )
      end
   until false
```

```
end;

procedure SDDriver;
begin
    AwakeSD;
    repeat
        receive( SD, PId, Op, Blck, Base );
        Encode( Blck, Cyl, Sec, Tr );
        SDNotify( Op, Cyl, Sec, Tr );
        send( SDController, true );
        receive( InterruptVector, SDReady );
        LookAt( Loc );
        Transfer( Loc, Base );
        send( PId, Base )
    until false
end;
```

L'azione realizzata dalla routine di interruzione é quella di risvegliare il driver che ha attivato un nuovo ciclo di operazioni del corrispondente controller che, nel caso del disco SD, é implementata supponendo che *SDReady* é la linea di connessione fra il controller e la relativa locazione del vettore delle interruzioni.

Sono state considerate predefinite le seguenti procedure: *Encode*, che codifica l'indirizzo logico *Blck* del blocco nella forma (Cyl, Sec, Tr); *SDNotify*, che trasferisce l'operazione nell'opportuno registro del controller; *LookAt*, che permette di ricavare l'indirizzo del buffer di transito fra controller e memoria centrale; *Transfer*, che permette di trasferire un blocco da/per la memoria centrale attraverso i canali di accesso diretto.

4.6 SEMAFORI PRIVATI

Una delle difficoltá che si presenta nell'utilizzo dei semafori é quella relativa ai problemi di schedulazione per l'accesso alle risorse e che si riflette nella necessitá di poter programmare esplicitamente la politica di gestione della coda di processi connessa col semaforo che controlla l'impiego della risorsa in questione.

D'altra parte la programmabilitá esplicita della coda dei processi in attesa sul semaforo vanificherebbe il vantaggio dell'uso dei *semafori come astrazioni sulle code di processi.* Per superare questa ed altre difficoltá sono stati introdotti i **semafori privati** definiti come *semafori su cui un solo processo puó eseguire l'operazione* P[8] mentre tutti gli altri possono eseguire solamente operazioni *V*.

Come si puó intuire, c'é una certa corrispondenza fra *semafori privati* in ambito variabili condivise e *porte* in ambito scambio messaggi. Il vantaggio dei semafori privati é duplice; da una parte, semplifica l'implementazione dell'operazione P perché non é piú necessario gestire una coda di processi, dall'altra, proprio in virtú di questo fatto, é possibile programmare le code di processi come *code di semafori privati* dei processi per i quali si vuol programmare esplicitamente uno schedulatore di utilizzo di risorsa.

[8]che puó vedersi come il proprietario del semaforo

4.6.1 Implementazione dei Semafori Privati

Come é stato appena sottolineato, l'implementazione dei semafori privati risulta piú semplice in quanto il kernel non deve piú gestire code di processi ma deve soltanto notificare all'unico processo che puó essere bloccato sul semaforo se é in condizioni di continuare l'esecuzione. Nella pratica, dunque, é sufficiente sostituire nel descrittore del semaforo il campo *coda* con il campo *identificatore di processo*. Se il valore di questo campo é nullo allora il processo, per il quale il semaforo é privato, é in grado di continuare l'esecuzione; altrimenti, il valore non nullo di tale campo rappresenta l'indice di processo temporaneamente bloccato su quel semaforo privato.

```
type PrivSem = ↑ Descriptor;

    Descriptor = record
                        Val : integer;
                        Job : Process
                     end;

procedure PrivP( var S : PrivSem );
varPId : Process;
   Done : boolean;
begin GetPid( PId );
   send( PService, S, Pid );
   receive( Kernel, Done )
end;

procedure V( var S : PrivSem );
begin
   send( VService, S )
end;

procedure Kernel;
var PId : Process;
      S : PrivSem;
begin
   InitS;
   do true; receive( PService, S, PId ) → with S↑ do
      if ( Val > 0 ) then
         begin
            Val := Val - 1;
            send( PId, true )
         end else
         Job := PId;
   [] true; receive( VService, S ) → with S↑ do
      if Job = 0 then
            Val := Val + 1 else
         begin
            Pid := Job;
            Job := 0;
```

```
                **send**( PId, **true** )
            end
        enddo;
    end;
```

Per mettere in maggior evidenza il fatto che l'operazione P é vincolata ad essere effettuata su un semaforo privato, é stata ridenominata operazione PrivP. I descrittori dei semafori privati possono essere allocati nei descrittori di processo per i quali i semafori hanno questa caratteristica.

4.6.2 Gestione di Risorse Multiple

Si consideri, allora, il problema di gestire un numero M di istanze di una risorsa di tipo assegnato da parte di un *pool di server*, ciascuno dei quali é autorizzato a compiere solo un numero ristretto di operazioni fra quelle possibili secondo un indice di prioritá specificato come segue. Se N é il numero dei server del pool, l'indice di processo[9] ne stabilisce la prioritá nel senso che *minore é l'indice di processo maggiore é la sua prioritá.* Ciascun server puó accedere ad un'istanza di risorsa utilizzando il protocollo

```
...........
Request( Pid );
< uso della risorsa >;
Release ( Pid )
...........
```

dove *uso della risorsa* rappresenta il codice di una delle operazioni eseguibili dal processo Pid.

Il protocollo d'impiego delle M istanze di risorsa prevede l'impiego di *semafori privati* definiti all'interno di ciascun server in accordo alle seguenti considerazioni. Innanzittutto, lo stato dei server é descritto da un opportuno *contesto* nella cui definizione devono comparire lo stato del processo, il semaforo privato su cui il processo é in attesa, se *bloccato*, e l'indice della risorsa che gli é stata assegnata se *puó usarla*.

Le definizioni di tipo e le dichiarazioni di variabili che seguono esemplificano la situazione

```
const   MaxProc = N;
        MaxRes = M;

type  Process = 0 .. MaxProc;
      RIndex = 0 .. MaxRes;
      Status = ( Blocked, CanUse );

      Context =   record
                     State : Status;
                     Using : semaphore;
                     Which : RIndex
                  end;

      ContextTable = array[ Process ] of Context;
```

[9] compreso fra *0* e N

```
                InUse = array[ RIndex ] of boolean;

var   Count : integer;
      Len : integer;
      Free : InUse;
      PCB : ContextTable;
      Mutex : semaphore;
```

dove il vettore Free, indicizzato sulle istanze della risorsa, stabilisce quando l'istanza di risorsa considerata é disponibile mentre il vettore PCB, indicizzato sull'identificatore di processo, determina, in base a tutte le informazioni del *contesto* di ciascun processo, la politica di allocazione della risorsa richiesta.

Partendo dalle definizioni testé viste, é comodo realizzare le seguenti procedure di supporto, alcune delle quali impiegano i semafori privati di cui s'é detto,

```
procedure Select( var R : RIndex );
var Found : boolean;
begin
   R := 0;
   repeat
      R := R + 1;
      Found := Free[ R ]
   until Found
end;

function IsWaiting( Pid : Process ) : boolean;
begin
   with PCB[ Pid ] do
      IsWaiting := State = Blocked
end;

procedure Activate( var Pid : Process );
varFound : boolean;
begin
   Pid := 0;
   repeat
      Pid := Pid + 1;
      Found := IsWaiting( Pid )
   until Found
end;

procedure PCBInit( Pid : Process );
begin
   with PCB[ Pid ] do
      begin
         State := Blocked;
         Which := 0;
         Initial( Using, 0 )
```

```
        end
end;

procedure Assign( R : RIndex; Pid : Process );
begin
    with PCB[ Pid ] do
        begin
            State := CanUse;
            Which := R;
            V( Using )
        end
end;

procedure Suspend( Pid : Process );
begin
    with PCB[ Pid ] do
        State = Blocked
end;

procedure Detach( Pid : Process; var R : RIndex );
begin
    with PCB[ Pid ] do
        begin
            R := Which;
            Which := 0
        end
end;

procedure Wait( Pid : Process ); external;
begin
    with PCB[ Pid ] do
        P( Using )
end;
```

il cui significato puó riassumersi in quanto segue:

- Select, selezionare la prima risorsa disponibile utilizzando il vettore *Free*,
- Wait, pone il processo in attesa sul proprio semaforo privato se non puó essergli allocata alcuna risorsa,
- IsWaiting, verifica se un processo é in attesa di risorsa,
- Activate, attiva il processo a prioritá piú alta (minore é il Pid maggiore é la prioritá),
- Assign, per assegna la risorsa R al processo Pid,
- Detach, consente al processo Pid di rilasciare la risorsa R,
- Suspend, notifica la sospensione del processo Pid.

in modo tale da poter realizzare il protocollo di accesso alla risorsa precedentemente indicato.

Facendo riferimento alle variabili intere *Count* e *Len*, la prima utilizzata per tener traccia del numero di istanze di risorsa ancora disponibili, la seconda per indicare il numero di processi in attesa di una istanza di risorsa, non é difficile realizzare le procedure *Request* e *Release*.

```
procedure Request( Pid : Process );
varK : RIndex;
begin
    P( Mutex );
    if ( Count > 0 ) then
        begin
            Count := Count - 1;
            Select( K );
            Free[ K ] := false;
            Assign( K, Pid )
        end else
        begin
            Len := Len + 1;
        Suspend( Pid )
        end;
    V( Mutex );
    Wait( Pid )
end;

procedure Release( Pid : Process );
varK : Process;
        J : RIndex;
begin
    P( Mutex );
    Detach( Pid, J );
    if ( Len > 0 ) then
        begin
            Len := Len - 1;
            Activate( K );
            Assign( J, K )
        end else
        begin
            Count := Count + 1;
            Free[ J ] := true
        end;
    V( Mutex )
end;

procedure Init;
begin
    Initial( Mutex, 1 );
    Len := 0;
```

```
        for K := 1 to MaxProc do
            PCBInit( K );
        Count := MaxRes;
        for J := 1 to MaxRes do
            Free[ J ] := true
    end;
```

Com'era facile aspettarsi il numero di semafori che é stato necessario introdurre per risolvere correttamente il problema é pari ad N + 1 , essendo N il numero di processi server considerati.

4.7 REGIONI CRITICHE

Si é giá accennato che, sebbene i semafori possano essere utilizzati per programmare qualunque tipo di sincronizzazione, P e V si presentano come primitive non strutturate e, per questo, prone ad errori se utilizzate in modo generale senza schemi ben assestati. Ad esempio, l'esecuzione di una sezione critica deve iniziare sempre con un'operazione P e terminare con l'operazione V sullo stesso semaforo. Tralasciando una delle due, oppure, codificando P e V su semafori diversi, gli effetti sarebbero disastrosi in quanto la mutua esclusione non potrebbe piú essere garantita.

Inoltre, il solo uso dei semafori non é in grado di assicurare che non si sia dimenticata qualche istruzione relativa a variabili condivise giá definite; l'effetto che si ottiene é ancora un'implementazione non corretta della mutua esclusione.

Per ovviare a queste difficoltá é sufficiente introdurre un'istruzione strutturata che sia in grado di racchiudere nel corretto protocollo di mutua esclusione tutte le sezioni critiche della stessa classe. Le variabili condivise vengono esplicitamente raggruppate in **risorse**, implementate come shared type, con la condizione che ogni variabile condivisa puó comparire al piú in una risorsa.

Per accedervi si possono utilizzare solamente istruzioni che compaiono in regioni critiche che nominano quella risorsa. Ad esempio, se la risorsa R contiene le variabili v_1, v_2,...,v_n, aventi rispettivamente tipi T_1, T_2,...,T_n, allora questa va dichiarata come record condiviso

```
type  Resource =  shared
                      record
                          v1 :T1;
                          v2 :T2;
                          ............
                          vn :Tn;
                      end

var   R :  Resource;
```

e ad essa vi si puá accedere solamente per mezzo di istruzioni che nominano R nella forma seguente

region R **do** S

dove S rappresenta una lista arbitraria di istruzioni. Diventa, allora, compito del compilatore introdurre l'appropriato semaforo di muta esclusione in corrispondenza di ciascuna dichiarazione di risorsa ed inizializzarlo correttamente ad uno. Inoltre le istruzioni region saranno delimitate dalle operazioni P e V sull'opportuno semaforo.

4.7.1 Ispezione e aggiornamento di una risorsa

Si consideri il problema di gestire una risorsa il cui accesso puó ottenersi per mezzo di due classi di operazioni, la prima di ispezione e la seconda di aggiornamento, le quali si presentano come sezioni critiche per i processi che le eseguono. Si supponga, allora, di controllare tali operazioni con un protocollo che soddisfa i seguenti vincoli:

- le operazioni di ispezione e di aggiornamento sono mutuamente esclusive;
- le operazioni di aggiornamento non si possono sovrapporre;
- non c'é alcun limite al numero di operazioni di ispezione che possono essere effettuate contemporaneamente.

La sua realizzazione puó ottenersi facilmente utilizzando il costrutto di regione critica semplice e i semafori secondo il seguente schema

....	....
Open(r)	**P**(w)
< ispeziona >	< aggiorna >
Close(r)	**V**(w)
....	....

dove r é una variabile intera condivisa[10] mentre le operazioni *Open* e *Close* sono realizzate dal segmento di codice riportato di seguito

```
type Gate = shared integer

procedure Open( var K : Gate );
begin
  region K do
    begin
      K := K + 1;
      if (K = 1) then P(w);
    end
end;

procedure Close( var K : Gate );
begin
  region K do
    begin
      K := K - 1;
      if (K = 0) then V(w);
    end
end;
```

nel quale la regione critica gioca il ruolo di meccanismo di controllo per le operazioni di ispezione mentre il semaforo serve a soddisfare le condizioni dei primi due punti.

La soluzione proposta puó apparire, a prima vista bizzarra, per il modo spregiudicato di introdurre operazioni di semaforo all'interno di sezioni critiche che, comé noto, sono esse stesse controllate da semafori di mutua esclusione. In particolare, l'operazione P sembrerebbe prona a produrre inevitabili situazioni di stallo.

Vogliamo, invece, provare attraverso l'invariante di semaforo e considerando le restrizioni sulla sequenza delle operazioni, imposte dalla forma delle procedure implementate, che il protocollo realizzato soddisfa le richieste. In particolare, detti **s** e **t**, rispettivamente, il numero dei processi che eseguono operazioni di aggiornamento e di ispezione, si vuole dimostrare che le due diseguaglianze seguenti

$$\begin{aligned} t = 0 &\implies 0 \le s \le 1 \\ t > 0 &\implies s = 0 \end{aligned}$$

[10] che prende il nome di *Gate* nell'implementazione

sono sempre verificate. Per provare la loro validità procediamo come nel caso dei semafori definendo, in particolare, le seguenti grandezze:

- opened(r), per contare il numero di operazioni *Open* completate, relativamente al Gate r;
- closed(r), per contare il numero di operazioni *Close* completate, riferite al Gate r;
- incs(r), per tener traccia del numero di incrementi positivi[11] della variabile r;
- decs(r), per registrare i decrementi positivi[12] della variabile r.

In questo modo é immediato scrivere le relazioni seguenti

$$\begin{aligned} opened(r) &\equiv incs(r) + received(w) \\ closed &\equiv decs(r) + sent(w) \\ received(w) &\leq sent(w) + 1 \end{aligned}$$

l'ultima delle quali rappresenta l'invariante per il semaforo w. Dalle precedenti discende, tenendo presente che la differenza fra *opened*(r) e *closed*(r) rappresenta il numero di processi che eseguono operazioni di ispezione

$$t \equiv opened(r) - closed(r) = received(w) - sent(w) + incs(r) - decs(r)$$

da cui deriva facilmente che le operazioni sul semaforo w eseguite dai processi ispettori[13] sono legate dalla relazione

$$\begin{aligned} received(w) - sent(w) &= t - [incs(r) - decs(r)] \\ &= t - [t - pos(t)] = pos(t) \end{aligned}$$

avendo definito la funzione pos nel seguente modo

$$pos(t) \equiv \begin{cases} t-1, & \text{se t} > 0 \\ 0, & \text{se t} = 0 \end{cases}$$

per cui gli incrementi di r eccedono i decrementi della quantitá *t - 1* se t > 0, altrimenti sono uguali. A questo punto, tenendo conto delle possibili operazioni di aggiornamento in atto, si puó scrivere

$$received(w) - sent(w) = s + pos(t)$$

che, introdotta nell'invariante di semaforo per w, dá luogo a

$$sent(w) \geq received(w) - 1 = sent(w) + s + pos(t) - 1$$

e, quindi, in definitiva,

$$s \leq 1 - pos(t)$$

da cui seguono immediatamente, tenendo presente la definizione di *pos* e il fatto che $s \leq 0$, le diseguaglianze viste in precedenza.

[11] nel senso che sono esclusi gli incrementi che portano il valore di r ad 1

[12] sono esclusi i decrementi che portano il valore di r a 0

[13] lettori

4.7.2 Regolazione del Traffico Cittadino

Si consideri il problema di regolare il flusso del traffico cittadino in modo che sia rispettata l'usuale regola della precedenza ossia, ad *ogni incrocio, le autovetture con diritto di precedenza continuano a fluire senza limite, mentre le altre possono occupare l'incrocio, una per volta, quando questo é libero.* Si supponga, in particolare, di dover definire gli appropriati protocolli di comportamento nella situazione in cui quattro strade a senso unico si incrociano ad angolo retto e su ciascuna di esse puó fluire un arbitrario numero di autovetture.

Poiché ciascun incrocio puó essere impegnato da una sola autovettura per volta, questo va trattato come una *regione critica* che ne controlla l'attraversamento[14] con l'ausilio di un semaforo supplementare per la regolazione del flusso senza diritto di precedenza. Inoltre, la strada che connette due incroci adiacenti, potendo contenere al piú N autovetture, deve essere trattata come una risorsa disponibile in N risorse fra di loro equivalenti.

Supponendo di impiegare, analogamente all'esempio precedente, il costrutto di *regione critica* semplice e i semafori, si deve

- definire, innanzitutto, la risorsa *Square* che rappresenta l'incrocio fra due strade;
- specificare, per ciascun incrocio, il semaforo ausiliario necessario a differenziare il flusso con diritto di precedenza da quello che non ce l'ha;
- implementino le procedure

  Enter(**var** Q : Square; **var** q : **semaphore**);
  Exit(**var** Q : Square; **var** q : **semaphore**)

  che definiscono correttamente il protocollo d'accesso all'incrocio per i flussi con diritto di precedenza.

Successivamente, si introduce per ciascuna strada che connette due incroci adiacenti l'ulteriore semaforo che controlla il numero di autovetture che possono essere contenute in ciascuna strada il cui protocollo d'impiego é dato dalle procedure

Arrive(**var** c : **semaphore**);
Advance(**var** w, c : **semaphore**);
Depart(**var** Q : Square; **var** w, r : **semaphore**);

per realizzano, rispettivamente, l'ingresso nel primo incrocio, l'accesso alla strada di connessione e l'uscita dal secondo incrocio.

Si tenga presente che ciascuna autovettura[15] non puó abbandonare la regione critica incrocio fino a quando non abbia verificato la disponibilitá di una istanza di risorsa che rappresenta la sua occupazione della strada in cui vi rimane fino a quando non é in grado di impegnare il secondo incrocio. Il codice riportato di seguito implementa uno dei quattro flussi

type Square = **shared integer**;

Flow = **record**

[14] realizzato dalla procedura primitiva *Cross*
[15] simulata con un processo

```
                    Common : Square; { secondo incrocio }
                    Access : semaphore; { secondo incrocio }
                    Street : semaphore; { strada fra gli incroci }
                    Before : semaphore; { primo incrocio }
                      end;

procedure Cross; external;
procedure Drive; external;

procedure Enter( var Q : Square; var q : semaphore );
begin
  region  Q do
    begin
      Q := Q + 1;
      if ( Q = 1 ) then P( q )
    end
end;

procedure Exit( var Q : Square; var q : semaphore );
begin
  region  Q do
    begin
      Q := Q - 1;
      if ( Q = 0 ) then V( q )
    end
end;

procedure Arrive( var c : semaphore );
begin
  P( c )
end;

procedure Advance( var w, c : semaphore );
begin
  P( w )
  V( c )
end;

procedure Depart( var Q : Square; var w, r : semaphore );
begin
  Enter( Q, r );
  V( w )
end;

procedure GoThrough( var F : Flow );
begin
  with F do
```

```
        begin
            Arrive( Before );
            Cross;
            Advance( Street, Before );
            Drive;
            Depart( Common, Street, Access );
            Cross;
            Exit( Common, Access );
        end
    end;
```

avendo supposto di poter disporre dell'operazione primitiva *Drive* che definisce l'avanzamento lungo la strada.

4.8 REGIONI CRITICHE CONDIZIONALI

Nella sezione precedente si sono visti alcuni esempi di sincronizzazione condizionale con l'uso esplicito di semafori che, peró, ne rendono difficile l'impiego in quanto le primitive su di essi sono le stesse sia che si tratti di mutua esclusione sia sincronizzazione condizionale. Questo rende difficile identificare lo scopo di un'operazione P o V senza considerare le altre operazioni fatte sullo stesso semaforo. Poiché la *sincronizazione condizionale* e la *mutua esclusione* sono concetti distinti, essi dovrebbero essere denotati con notazioni distinte.

Le *regioni critiche condizionali* rappresentano una notazione strutturata per specificare la sincronizzazione. Le variabili condivise, come nel caso delle *regioni critiche semplici*, vengono raggruppate in **risorse** in cui ciascuna variabile compare al piú in una risorsa. Ad esse vi si accede solamente per mezzo di istruzioni che compaiono in regioni critiche condizionali che nominano quella risorsa, le quali possono contenere espressioni booleane per fornire un meccanismo esplicito di sincronizzazione condizionale.

La risorsa R contenente le variabili v_1, v_2,...,v_n, aventi rispettivamente tipi T_1, T_2,...,T_n, viene dichiarata come

```
type  Resource =  shared
                     record
                        v1 :T1;
                        v2 :T2;
                        ............
                        vn :Tn;
                     end

var   R  :  Resource;
```

e su di essa si possono eseguire istruzioni, che fanno riferimento ad R, solamente nella forma seguente

region R **when** B **do** S

dove B é un'espressione booleana mentre S rappresenta una lista di istruzioni.

Si voglia, ad esempio, implementare i semafori con un costrutto di *regione critica condizionale*. Si definisce, innanzitutto, il tipo *Semaphore* come shared integer e, poi, le primitive di accesso ai semafori P e V con il codice seguente

```
type  Semaphore = shared integer;

procedure P( var S : Semaphore );
begin
   region S when ( S > 0 ) do S := S - 1
end;

procedure V( var S : Semaphore );
begin
   region S do S := S + 1
end;
```

senza introdurre esplicitamente la coda associata al semaforo che, ora, viene gestita implicitamente dall'implementazione della *regione critica condizionale*.

Un altro esempio, che ammette una soluzione relativamente semplice, é quello che si riferisce al problema del produttore/consumatore. Anche in questo caso si tratta di dare una definizione esplicita del tipo *Buffer*, che realizza la comunicazione tra i due processi, e su di essa implemetare le operazioni di inserzione e prelievo.

```
const BufferSize = N;

type  Index = 0 .. BufferSize;
      Buffer = record
                   Data : array[ Index ] of Item;
                   Size : Index;
                   Loc : Index;
                   Pos : Index;
               end;

procedure Deposit(var Bu : Buffer; Msg : Item );
begin
   region Bu when ( Size < BufferSize ) do
      begin
         Size := Size + 1;
         Data[ Pos ] := Msg;
         Pos := ( Pos + 1 ) mod BufferSize
      end
end;

procedure Fetch(var Bu : Buffer; var Msg : Item );
begin
   region Bu when ( Size > 0 ) do
      begin
         Size := Size - 1;
         Msg := Data[ Loc ];
         Loc := ( Loc + 1 ) mod BufferSize
      end
end;
```

4.8.1 Attraversamento di un Ponte a Senso Unico

Si consideri ora il problema dell'attraversamento di un ponte che permette il passaggio degli autoveicoli a senso unico alternato. Indicati con *North* e *South* le due sezioni di strada, a doppio senso di marcia, cui il ponte é connesso, il costrutto di *regione critica condizionale* permette di definire agevolmente un meccanismo di sincronizzazione per l'accesso al ponte, mediante l'introduzione di un'opportuna risorsa di tipo *Bridge*.

```
type CrossBridge = shared record
                        Southern : integer;
                        Northern : integer
                     end;

var    Bridge : CrossBridge;

procedure SouthCar;
begin
   region Bridge when ( Northern = 0 ) do
      Southern := Southern + 1;
   { attraversa il ponte }
   region Bridge do
      Southern := Southern - 1
end;

procedure NorthCar;
begin
   region Bridge when ( Southern = 0 ) do
      Northern := Northern + 1;
   { attraversa il ponte }
   region Bridge do
      Northern := Northern - 1
end;
```

Si noti come nella soluzione proposta i due flussi di autovetture vengono controllati senza alcun privilegio per la direzione di provenienza[16], potendo accedere e attraversare il ponte senza conflitti di direzione.

4.8.2 Implementazione

Benché il costrutto di regione critica condizionale abbia la capacitá di esprimere in modo molto coinciso molti schemi di programmazione concorrente, si presenta molto costoso da implementare per il fatto che le espressioni condizionali B, che controllano l'esecuzione delle istruzioni associate, possono contenere riferimenti a variabili locali, costringendo la rivalutazione di B per ogni processo in attesa. Infatti, la presenza di variabili locali non permette di affermare che se il primo processo trova la condizione falsa, questa é falsa per tutti gli altri processi bloccati sulla stessa condizione.

Comunque, anche l'implementazione piú banale deve costringere i processi ad una rivalutazione continua della condizione su cui sono temporaneamente bloccati, fino a quando non la trovano verificata. In questo senso, il segmento di codice riportato di seguito potrebbe risolvere il problema

[16] e, quindi, fornendo una soluzione simmetrica per i due flussi di traffico

```
var Found : boolean;

Found := false;
repeat;
  region R do
    begin
      S; Found := true
    end
until Found
```

se non fosse per il fatto che costringe i processi a rimanere in una situazione di *busy-waiting*.

L'implementazione effettiva, invece, dovrá bloccare i processi che trovano la condizione falsa secondo lo schema seguente

```
P(Mutex)
while not B do
  begin
    Jobs := Jobs + 1;
    if Next > 0 then
      V(Turn) else
      V(Mutex);
    P(Cond);
    Jobs := Jobs - 1;
    Next := Next + 1;
    if Jobs > 0 then
      V(Cond) else
      V(Turn);
    P(Turn);
    Next := Next - 1;
  end;
S;
if Jobs > then
  V(Cond) else
  if Next > 0 then
    V(Turn) else
    V(Mutex);
```

avendo indicato con *Mutex* il semaforo di mutua esclusione per l'accesso alla regione critica condizionale[17] mentre i semafori *Cond* e *Turn*, entrambi inizializzati a zero, il primo utilizzato per gestire la coda dei processi che trovano la condizione falsa e che, successivamente, vengono spostati su *Turn* per poterla rivalutare. Una soluzione semplificata si puó ottenere nell'ipotesi che la politica di gestione delle code dei semafori sia di tipo FIFO.

In ogni caso é immediato rendersi conto come in un sistema multiprogrammato la valutazione della condizione booleana B dia luogo a numerosi scambi di contesto[18] molti dei quali sono improduttivi perché il processo riattivato puó ritrovare ancora falsa la condizione B.

---

[17] inizializzato ad 1

[18] vale a dire,frequenti salvataggi e ripristini dello stato dei processi