Capitolo 5

MONITOR

Le regioni critiche condizionali sono costose da realizzare su processori singoli; inoltre, le istruzioni che eseguono operazioni sulle variabili che rappresentano risorse sono disperse fra i processi che le utilizzano. Questo significa che si deve analizzare il codice dell'intero programma concorrente per vedere tutti i modi con cui la risorsa é utilizzata. I monitor alleviano entrambe queste deficienze *incapsulando sia la definizione della risorsa che le operazioni che la manipolano*. In questo modo é possibile vedere una risorsa soggetta ad accessi concorrenti come un modulo consentendo al programmatore puó ignorare i dettagli implementativi della risorsa quando la usa e di ignorare come usarla quando programma il monitor che la implementa.

Un **monitor** *consiste di una collezione di* variabili permanenti, *usate per rappresentare lo stato di una risorsa, e da alcune procedure che implementano le operazioni su di essa.*

Il monitor é costituito anche da un codice di *inizializzazione delle variabili permanenti*, che viene eseguito una volta sola, prima dell'esecuzione di qualunque procedura. I valori delle variabili permanenti sono mantenuti fra attivazioni distinte di procedure di monitor e sono accessibili soltanto da esse per il tramite del monitor.

Le procedure di monitor possono avere parametri e variabili locali, ciascuna delle quali prende un nuovo valore ogni volta che la procedura viene attivata. La struttura sintattica di un monitor, avente nome *MonName*, e procedure Op_1, Op_2,...,Op_n, è quella riportata di seguito

type MonName = **monitor**;
    var {*dichiarazione delle variabili permanenti*};
    procedure entry Op_1(< *parametri* >);
        var {*dichiarazione variabili locali*};
        begin
            {*codice che implementa* Op_1}
        end;
    ....
    procedure entry Op_n(< *parametri* >);
        var {*dichiarazione variabili locali*};
        begin
            {*codice che implementa* Op_n}
        end;
    begin
        {*codice di inizializzazione delle variabili permanenti*};
    end

La procedura Op_j definita all'interno del monitor *MonName* viene invocata eseguendo l'istruzione

$$Mon.Op_j(< argomenti >)$$

essendo *Mon* una viariabile di tipo monitor dichiarata tramite il costrutto

$$\textbf{var}\ Mon : MonName$$

il cui effetto é quello di allocare spazio per *Mon* secondo le direttive del monitor e inizializzarne il valore secondo il codice di inizializzazione definito all'interno del monitor.

Per quanto riguarda, invece, il controllo sull'ordine di accesso alle variabili del monitor sono stati proposti una varietà di costrutti linguistici i quali permettono di classificare i monitor in due classi

- monitor con primitive di sincronozzazione *esplicite*;
- monitor con sincronozzazione condizionata *implicita*

Nel primo caso si hanno, essenzialmente, quattro tipi di monitor, ciascuno dei quali è caratterizzato sia delle primitive di sincronizzazione che dalle variabili su cui tali primitive agiscono. Si ha, pertanto, il seguente schema

	tipo di monitor	variabile di sincronizzazione	operazione di sincronizzazione
(a)	Hoare	condition	wait/signal
(b)	Mitchell	condition	wait/notify
(c)	Brinch-Hansen	queue	delay/continue
(d)	Wirth	signal	wait/send

Nel secondo caso, invece, giocano un ruolo fondamentale le espressioni booleane che determinano la sincronizzazione

	tipo di monitor	variabile di sincronizzazione	operazione di sincronizzazione
(d)	Hoare	....	wait($<$ *espressione* $>$)
(e)	Kessel	....	await($<$ *variabile* $>$)

dove $<$ *variabile* $>$ é un riferimento ad un'espressione booleana che puó contenere solamente variabili di monitor.

5.1 MONITOR DI HOARE

Per ritardare l'esecuzione dei processi all'interno del monitor di Hoare si devono dichiarare un certo numero di variabili di tipo condition a cui si puó accedere solamente per mezzo delle operazioni primitive *wait/signal*.

Se p é una variabile di tipo *condition* e, quindi, dichiarata all'interno del monitor con il costrutto

$$\textbf{var}\ p : \textbf{condition};$$

l'operazione

- p.**wait** fá si che il processo che l'ha invocata si sospenda su p rilasciando il controllo esclusivo del monitor;
- p.**signal** fá si che il processo che l'ha invocata continui l'esecuzione se non c'é alcun processo bloccato su p; altrimenti il processo viene temporaneamente sospeso mentre uno dei processi bloccati su p viene riattivato.

Il processo che si sospende, in seguito all'operazione p.**signal**, riprende l'esecuzione solamente quando non c'é nessun altro processo in esecuzione nel monitor. Inoltre, tali processi[1] hanno una prioritá maggiore su ogni processo che tenta di iniziare l'esecuzione di procedura del monitor. Naturalmente si suppone che le variabili di tipo *condition* godano della proprietá dell'equitá[2] nel senso che *i processi sospesi su una variabile condizione, segnalati infinitamente spesso, non possono rimanere sospesi per sempre.*

Si noti che la presenza di variabili di tipo *condition* consente la presenza di piú processi all'interno del monitor anche se tutti, tranne uno, sono ritardati da operazioni *wait* o *signal*.

La realizzazione del monitor di Hoare si ottiene introducendo un certo numero di semafori e variabili in accordo al seguente schema.

- Si introduce un semaforo *Turn* e una variabile *Jobs* per ciascuna variabile di tipo condition.
- Si definiscono i semafori *Mutex* e *Urgent* per controllare l'accesso al monitor da parte dei processi che ne fanno richiesta, introducendo l'ulteriore variabile *Count* che tiene conto delle prioritá di accesso.

Si ricordi che l'implementazione dell'operazione *cond* .**signal** deve tener presente che il processo che l'esegue, dovendo segnalare ad un processo in attesa che puó continuare, rilascia il controllo del monitor. Per quanto riguarda, invece, la struttura di ciascuna procedura di monitor visibile dall'esterno é sufficiente indicare in che modo viene realizzata la mutua esclusione. Si ha, pertanto, lo schema seguente

```
type Condition = record
        Jobs : integer;
        Turn : semaphore
    end;

var Mutex : semaphore;
        Urgent : semaphore;
        Count : integer;

procedure Entry( procedure Proc );
begin
  P( Mutex );
  Proc;
  if Count = 0 then
    V( Mutex ) else
    V( Urgent )
end;
```

[1] segnalanti
[2] fearness

```
procedure Wait( var Cond : Condition );
begin
  with Cond do
    begin
      Jobs := Jobs + 1;
      if ( Count > 0 ) then
        V( Urgent ) else
        V( Mutex );
      P( Turn );
      Jobs := Jobs - 1
    end
end;

procedure Signal( var Cond : Condition );
begin
  with Cond do
    if Jobs > 0 then
      begin
        Count := Count + 1;
        V( Turn );
        P( Urgent );
        Count := Count - 1
      end
end;
```

E' facile rendersi conto che le variabili di tipo *condition* e i semafori hanno un certo numero di affinitá. Vale la pena osservare, peró, che l'operazione signal, contrariamente alla sua omologa V sui semafori, non lascia traccia nel senso che un processo ritarda sempre dopo aver eseguito una wait, anche se una precedente operazione signal non ha risvegliato alcun processo.

Questa proprietá rende difficile l'uso corretto delle operazioni signal e wait in quanto devono essere utilizzate altre variabili per ricordare l'esecuzione di un'operazione signal. Inoltre, queste variabili devono essere analizzate prima di eseguire una wait per guardarsi dall'attendere che l'evento corrispondente ad una signal si sia giá verificato.

Inoltre, rispetto alle regioni critiche condizionali, non è prevista alcuna associazione sintattica fra espressioni booleane di controllo e le corrispondenti operazioni signal, wait o alle stesse variabili di tipo *condition*. Pertanto, non é facile determinare *perchè un processo é stato ritardato* su una variabile condizionale a meno che le operazioni signal e wait non vengano utilizzate in modo molto disciplinato:

- ogni operazione wait su una variabile *condizionale* p deve essere contenuta in un'istruzione **if**

  if not B_1 **then** p.**wait**

  in cui l'espressione booleana B_1 rappresenta la negazione della condizione di sincronizzazione desiderata;

- ogni operazione signal sulla stessa variabile condizionale deve essere contenuta in una istruzione condizionale

  if B_2 **then** p.**signal**

in cui l'espressione booleana B_2 dá la condizione di sincronizzazione voluta.

Sintatticamente, *identiche espressioni booleane* possono avere valori differenti se contengono referenze a variabili locali. Cosí non c'é alcuna garanzia che un processo risvegliato veda realmente la condizione per cui stava attendendo.

Infine, poiché l'operazione signal é espropriante, lo stato delle variabili permanenti visto dal processo che l'ha eseguito puó cambiare nell'intervallo di tempo alla fine del quale il processo riprende l'esecuzione.

5.1.1 Gestione di una Stampante

Si consideri il problema di realizzare la gestione di una stampante il cui driver utilizza una ipotetica primitiva *Put*(Bu) che scrive il contenuto di un buffer come linea di stampa. Utilizzando il costrutto di monitor appena visto vanno definiti

- la struttura dati necessaria a sostenere le operazioni del caso,
- la procedura Request che, invocata dal cliente *ClientId*, ha l'effetto di servire solo le proprie richieste successive, ritardando tutte le richieste degli altri clienti,
- la procedura Release, che permette a *ClientId* di rilasciare la stampante,
- la procedura di stampa Print.

Tenendo presente lo schema generale del monitor, nel caso in esame si istanzia secondo il codice riportato di seguito.

```
type LinePrinter = monitor;

   var   Locked : boolean;
         Curr : ClientId;
         Jobs : integer;
         Lpr : condition;

   procedure entry Print( PID : ClientId, Buf : Buffer );
   begin
      if (not Locked) or
            ( PID <> Curr ) and ( Jobs > 0 ) then
            begin
               Jobs := Jobs + 1;
               Lpr.wait;
               PutBuf( Buf );
               Jobs := Jobs - 1;
               Lpr.signal;
            end else
            PutBuf( Buf )
   end;

   procedure entry Print( PID : ClientId, Bu : Buffer );
   begin
```

```
        if ( Locked and ( PID = Curr )) or
                ( not Locked and ( Jobs = 0 )) then
            Put( Bu ) else
            begin
                Jobs := Jobs + 1;
                Lpr.wait;
                Put( Bu );
                Jobs := Jobs - 1;
                Lpr.signal
            end
    end;

    procedure entry Request( PID : ClientId );
    begin
        if Locked then
            begin
                Jobs := Jobs + 1;
                Lpr.wait
            end;
            Locked := true;
            Curr := PID
    end;

    procedure entry Release;
    begin
        Locked := false;
        Jobs := Jobs - 1;
        Lpr.signal
    end;

begin
    Locked := false;
    Jobs := 0
end;
```

5.1.2 Il Problema del Barbiere Assonnato

La bottega di un barbiere comprende una stanza d'attesa con N sedie e l'angolo di taglio in cui é sistemata l'apposita poltrona. Si consideri l'attivitá del barbiere cosí schematizzata.

- Se non ci sono clienti il barbiere va a riposare.
- Se un cliente entra nella bottega e trova tutte le sedie occupate, allora rinuncia al taglio di capelli.
- Se il barbiere é impegnato con un altro cliente, allora attende il proprio turno su una delle sedie disponibili.
- Se il barbiere sta riposando, allora il cliente lo sveglia.

Tenendo presente il costrutto di monitor, con il citato meccanismo di sincronizzazione, la soluzione si presenta nel modo seguente

```
type BarberShop = monitor;

    var   Chair : 0 .. N + 1;
          Seats : integer;
          Bell : condition;
          Turn : condition;

    function entry CutRequest : boolean;
    var Sitting : boolean;
    begin
        Sitting := Chair <= Seats;
        if Sitting then
            begin
                Chair := Chair + 1;
                if ( Chair > 1 ) then Turn.wait;
            end;
        CutRequest := Sitting
    end;

    procedure entry ExitShop;
    begin
        if ( Chair = 1 ) then Bell.signal
    end;

    procedure entry CutService;
    begin
        Chair := Chair - 1;
        if ( Chair = 0 ) then Bell.wait
    end;

    procedure entry Continue;
    begin
        if ( Chair > 2 ) then Turn.signal
    end;

begin
    Chair := 0
end;
```

L'operazione *CutRequest* é effettuata da ogni cliente che desidera avere un taglio di capelli e restituisce il valore **true** se esistono posti disponibili; l'operazione *CutService* é effettuata dal barbiere su richiesta del cliente quando é giunto il proprio turno.

Le procedure di monitor sono seguite, rispettivamente, da due azioni primitive *SitDown* e *HairCut*, la prima avente il significato di occupazione della sedia del barbiere, la seconda di esecuzione del taglio di capelli. Le ulteriori operazioni ExitShop, eseguita dal cliente, e Continue, eseguita dal barbiere, determinano l'ovvio protocollo eseguito da un cliente e dal barbiere.

5.1.3 Gestione di una Risorsa con Prioritá

Si consideri un sistema di calcolo in cui sono presenti N processi P_1, P_2,..., P_N che condividono una unica risorsa R utilizzabile, al piú, da un processo alla volta. Se la risorsa é libera, un processo puó iniziare l'esecuzione usandola immediatamente secondo lo schema:

```
process Pi;
repeat
    reserve( i );
    use resource;
    release
forever;
```

In caso contrario, il processo deve attendere finché la risorsa non viene rilasciata da un altro processo. Se uno o piú processi stanno aspettando il rilascio della risorsa, questa viene assegnata al processo con piú alta prioritá accordo alla seguente regola: *il processo P_i ha prioritá* i*, con $1 \leq i \leq N$, i cui valori piú bassi indicano prioritá maggiore.*

L'implementazione delle procedure di accesso alla risorsa sono dichiarate come procedure di monitor nel quale vanno definite anche le opportune variabili e dichiarazioni di tipo. La soluzione é quella riportata di seguito.

```
type Resource = monitor;

    const MaxProc = N;

    type Process = 1 .. MaxProc;
            Queue = array[ Process ] of condition;
            Event = array[ Process ] of boolean;

    var Free : boolean;
        Qu : Queue;
        Turn : Event;
        K : Process;

    procedure entry Reserve( Pid : Process );
    begin
        if Free then
            Free := false else
            begin
                Turn[ Pid ] := true;
                Qu[ Pid ].wait
            end
    end;

    procedure entry Release;
```

```
    var Done : boolean;
        K : Process;
    begin
            Done := false;
            K := 0;
            repeat
                K := K + 1;
                if Turn[ K ] then
                    begin
                        Turn[ K ] := false;
                        Qu[ K ].signal;
                        Done := true
                    end
            until ( K = MaxProc ) or Done;
            if not Done then Free := true
    end;

begin
    Free := true;
    for K := 1 to MaxProc do Turn[ K ] := false
end;
```

In essa, i tipi *Queue* e *Event* sono indiciati rispetto al nome dei processi, in modo tale che ciascun abbia associata una variabile condizione su cui attendere lo sblocaggio della risorsa ed un indicatore che tiene traccia di tale attesa.

5.1.4 Il Problema del Same Fringe

Si supponga di dover realizzare un algoritmo che verifichi quando due alberi assegnati hanno le stesse foglie, senza per questo essere identici[3]. La soluzione piú immediata si ottiene, come é facile rendersi conto, costruendo per ciascun albero la lista delle foglie che lo costituiscono secondo un certo ordine di attraversamento[4], e quindi, accertarsi che le due liste siano uguali. E' presto visto che tale soluzione non é desiderabile perché anche quando le prime foglie sono diverse é necessario, comunque, attraversare i due alberi per costruire le due liste, prima di verificare che queste differiscono a partire dai primi elementi.

Per superare tale inconveniente é comodo riscrivere questo algoritmo in forma parallela facendo in modo che la generazione delle due liste di foglie[5] proceda contemporaneamente al controllo della loro eguaglianza.

L'impiego del costrutto di monitor appena visto permette di realizzare elegantemente la funzione

$$\mathsf{SameFringe}(T_1, T_2 : \mathit{Tree}) : \textbf{boolean};$$

la quale ha lo scopo di generare due processi, attivati eseguendo un'unica procedure *Flatten* che prende in ingresso un albero e produce, in uscita, la lista delle sue foglie; l'attivazione di un terzo processo,

[3] o, come si dice, abbiano la stessa frontiera
[4] tipicamente, top-down depth-first
[5] una, per ciascun albero

ottenuto dall'esecuzione della procedura *Compare*, verifica che gli elementi omologhi delle due liste siano uguali. Non appena viene individuata una coppia di elementi distinti, i processi corrispondenti alla procedura Flatten vengono bloccati, il processo che esegue il confronto restituisce il valore falso alla funzione SameFringe e termina.

Secondo queste specifiche, l'implementazione della funzione SameFringe è ottenuta definendo le variabili comuni ai tre processi come *monitor* e realizzando le due procedure Flatten e Compare. Il codice di SameFringe deve prevedere, inoltre, la loro attivazione e la loro terminazione secondo le specifiche assegnate.

Per quanto riguarda la struttura dati *Tree*, si supponga che questa sia giá stata realizzata come una struttura astratta di dati con le seguenti procedure predefinite

> **procedure** Cons(T_1, T_2 : Tree; **var** T : Tree); **external**;
> **procedure** Car(T_1 : Tree; **var** T : Tree); **external**;
> **procedure** Cdr(T_2 : Tree; **var** T : Tree); **external**;
> **function** Eq(L_1, L_2 : Leaf) : **boolean**; **external**;
> **function** Null(T : Tree) : **boolean**; **external**;
> **function** Atom(T : Tree) : **boolean**; **external**;

il significato di ciascuna essendo, rispettivamente,

- dati due alberi, costruire un nuovo albero in cui T_1 e T_2 sono, rispettivamente, i sottoalberi sinistro e destro;
- dato un albero, estrarre il sottoalbero sinistro;
- dato un albero, estrarre il sottoalbero destro;
- dati due nodi terminali, verificare che siano gli stessi;
- dato un albero, verificare che sia nullo;
- dato un albero, verificare che sia un nodo terminale

Ovviamente, si suppone che gli alberi siano binari ed, inoltre, *Leaf* é una struttura predefinita che rappresenta i nodi terminali degli alberi.

Con queste ipotesi la soluzione si presenta nel modo seguente

```
type Position = ( First, Second );

        Buffer = monitor

    type Index = 0 .. BufferSize - 1;
        Size = 0 .. BufferSize;
        Datum = record
                One : Leaf;
                Two : Leaf;
                How : integer
            end;
        BufferData = array[ Index ] of Datum;

    var List : BufferData;
```

```
Dim : Size;
LC1 : Index;
LC2 : Index;
Curr : Index;
NotEmpty : condition;
NotFull : condition;

procedure InitB;
var K : Index;
begin
    LC1 := 0;
    LC2 := 0;
    Curr := 0;
    for K := 0 to BufferSize - 1 do List[ K ].How := 0
end;

function PutB( L : Leaf; Pos : Position ) : boolean
begin
    case Pos of
        First : begin
            with List[ LC1 ] do
                begin
                    One := L;
                    How := How + 1;
                    PutB := ( How = 2 )
                end;
            LC1 := ( LC1 + 1 ) mod BufferSize
            end;
        Second : begin
            with List[ LC2 ] do
                begin
                    Two := L;
                    How := How + 1;
                    PutB := ( How = 2 )
                end;
            LC1 := ( LC1 + 1 ) mod BufferSize
            end
end;

procedure GetB( var L1, L2 : Leaf );
begin
    with List[ Curr ] do
        begin
            L1 := One;
            L2 := Two;
            How := 0
        end;
```

```
            Curr := ( Curr + 1 ) mod BufferSize
        end;

    procedure entry Deposit( L : Leaf; Pos : Position );
    begin
        if ( Size = BufferSize ) then NotFull.wait;
        Size := Size + 1;
        if PutB( L, Pos ) then NotEmpty.signal
    end;

    procedure entry Fetch( var L1, L2 : Leaf );
    begin
        if ( Size = 0 ) then NotEmpty.wait;
        Size := Size - 1;
        GetB( L1, L2 );
        NotFull.signal
    end;

begin
    InitB;
    Size := 0;
end;


    SigKill = monitor;

    var StopF : boolean;

    function entry Which : boolean;
    begin
        Which := StopF
    end;

    procedure entry Kill;
    begin
        StopF := true
    end;

begin
    StopF := false
end;


function SameFringe( T1, T2 : Tree ) : boolean;
var CompF : boolean;
        Bu : Buffer;
        Stop : SigKill;
```

```
procedure Flatten( T : Tree; Pos : Position );
var Trail : Tree;
      Aux : Tree;
   HaltF : boolean;
begin
   Trail := nil;
   HaltF := false;
   repeat
      if Atom( T ) then Bu.Deposit( T, Pos )
      else if Null( T ) then
         if Null( Trail ) then
            begin
               HaltF := true;
               Bu.Deposit( nil, Pos )
            end else
            begin
               Car( Trail, T );
               Cdr( Trail, Trail )
            end else
            begin
               Cdr( T, Aux );
               Cons( Aux, Trail, Trail );
               Car( T, T )
            end;
      if not HaltF then HaltF := Stop.Which
   until HaltF
end;

procedure Compare;
var HaltF : boolean;
      L1 : Leaf;
      L2 : Leaf;
begin
   HaltF := false;
   repeat
      Bu.Fetch( L1, L2 );
      CompF := Eq( L1, L2 );
      if not CompF then
         begin
            HaltF := true;
            Stop.Kill
         end
   until HaltF
end;

begin
```

```
        cobegin
            Flatten( T1, First );
            Flatten( T2, Second );
            Compare
        coend;
        SameFringe := CompF
    end;
```

Come si può notare, il programma principale contiene solamente l'attivazione concorrente delle tre coroutine, l'ultima delle quali è in grado di produrre un segnale d'arresto per le altre due, utilizzando il monitor SigKill.

5.1.5 Gestione di una Base di Dati

Si consideri il problema di realizzare la gestione di una base di dati, condivisa da un certo numero di clienti, su cui vengono eseguite transazioni di vario tipo, implementate come procedure. Accanto alle necessarie strutture dati, si supponga che il controllo d'accesso del monitor sia realizzato dalle procedure

- Request che, invocata da un cliente *Pid*, ha l'effetto di servire solo le successive richieste di *Pid*, ritardando tutte le richieste degli altri clienti;
- la procedura Release, che permette a *Pid* di rilasciare il controllo della base di dati;
- la procedura di gestione delle transazioni *Update* in modo tale che esegua solamente le transazioni attribuibili allo stesso cliente purché la base di dati non sia disponibile.

La soluzione al problema proposto assume, allora, la forma seguente

```
    type DBManager = monitor;

        var   Locked : boolean;
              Curr : ClientId;
              Jobs : integer;
              DBM : condition;

        procedure entry Request( PID : ClientId );
        begin
            if Locked then
                begin
                    Jobs := Jobs + 1;
                    DBM.wait
                end;
                Locked := true;
                Curr := PID
        end;

        procedure entry Release;
        begin
            if Jobs = 0 then
```

```
            Locked := false else
            begin
                Jobs := Jobs - 1;
                DBM.signal
            end
    end;

    procedure entry UpDate( PID : ClientId; procedure Action );
    begin
        if (( Locked and ( PID = Curr )) or
                    ( not Locked and ( Jobs = 0 ))) then
            Action else
            begin
                Jobs := Jobs + 1;
                DBM.wait;
                Action;
                Jobs := Jobs - 1;
                DBM.signal
            end
    end;

begin
    Locked := false;
    Jobs := 0
end;
```

dove *Action* rappresenta una delle possibili transazioni che il cliente PID puó invocare.

5.2 SCHEDULAZIONE DI PROCESSI

In alcuni casi è auspicabile che il programmatore possa disporre di un maggior controllo sull'ordine con cui risvegliare i processi in attesa su una certa condizione. Per implementare questo tipo di *schedulazione a medio termine*[6] si puó utilizzare un'istruzione **wait** con *prioritá* nella forma seguente

$$p.\mathbf{wait}(t)$$

la cui semantica é esattamente la stessa dell'istruzione p.**wait** tranne per il fatto che i processi bloccati sulla variabile, di tipo *condition*, p sono risvegliati per ordine crescente di t. Appare chiaro che la proprietá di equitá non puó essere garantita a variabili *condition* usate in questo modo.

5.2.1 Shortest Job Next

Un problema molto comune che coinvolge la schedulazione di medio termine é quello che si riferisce a politiche di allocazione delle risorse note come: *il lavoro più breve prima*[7]. Si consideri, a questo

[6]medium-term scheduling, da non confondersi con lo *short-term scheduling*, che si riferisce al modo con cui assegnare i processi pronti ai processori disponibili, e il *long-term-scheduling*, che stabilisce il modo con cui selezionare i *lavori* da mandare in esecuzione

[7]shortest job next

proposito, il caso di una risorsa che puó essere allocata ad un solo processo per volta; se più processi sono in attesa del rilascio della risorsa, allora questa viene allocata a chi la userá per l'intervallo di tempo piú breve.

Il monitor riportato di seguito mostra come implementare lo schema di allocazione indicato.

```
type SJF = monitor;

    var   Free : boolean;
          Turn : condition;

    procedure entry Request( Time : integer );
    begin
        if not Free then
            Turn.wait( Time );
        Free := false;
    end;

    procedure entry Release;
    begin
            Free := true
            Turn.signal
    end;

begin
    Free := true
end;
```

Come si vede il monitor é costituito da due procedure: la prima, Request, che é chiamata dai processi che vogliono accedere alla risorsa per un intervallo di tempo *Time*; la seconda, Release, eseguita dai processi che rilasciano la risorsa.

5.2.2 Gestione di una Risorsa Multipla

In alcuni casi, rispetto all'esempio precedente, si può procedere senza utilizzare la prioritá come parametro di una wait. Si consideri, a questo proposito, il problema di gestire mediante uno scheduler una risorsa multipla caratterizzata, cioé, dal fatto di essere disponibile in un certo numero N di istanze], tra di loro tutte equivalenti. Si supponga che ciascun processo possa chiedere o rilasciare un numero R di istanze della risorsa, rispettivamente, mediante le procedure

```
procedure Allocate( R : integer );
procedure Release( B : integer );
```

le quali determinano l'ovvio protocollo d'utilizzo della risorsa. Se una richiesta di allocazione non puó essere soddisfatta, in quanto il numero di risorse disponibili é inferiore alla richiesta fatta dal processo, questo viene sospeso.

L'implementazione delle operazioni di allocazione e rilascio della risorsa é dato allora dal monitor riportato di seguito.

```
type Scheduler = monitor;

   var Count : integer;
      Awaited : integer;
      Free : condition;
      Qu : Queue;

   procedure entry Allocate( L : integer );
   begin
      while Count <= L do
         begin
            Awaited := Awaited + 1;
            Insert( Qu, L );
            Free.wait
            Remove( Qu, L );
            Awaited := Awaited - 1;
         end
      Count := Count - L
   end;

   procedure entry Release( L : integer );
   var K : integer;
   begin
      Count := Count + L
      if Awaited > 0 then Free.signal
   end;

begin
   Create( Qu );
   Count := N;
   Awaited := 0
end;
```

avendo utilizzato il semplice schema di sincronizzazione di Hoare e avendo indicato con L il numero di istanze della risorsa allocate oppure rilasciate. Si suppongano, inoltre, disponibili le seguenti operazioni

```
procedure Create( var Qu : Queue );
procedure Insert( var Qu : Queue; K : integer );
procedure Remove( var Qu : Queue; var K : integer );
```

la prima per inserire una richiesta nella coda delle richieste, la seconda per estrarne il primo elemento, tenendo presente che la coda di richieste é gestita nello stesso modo con cui é gestita la coda dei processi in attesa associati alle variabili di tipo condition del monitor.

5.2.3 Scansione Circolare dei Cilindri di un Disco

L'esempio che segue si riferisce al problema di schedulare il movimento delle testine di un disco in modo da minimizzare i tempi d'accesso che, nel nostro caso, si traduce nel ridurre al minimo i movimenti della testina. Si considerino, allora, le richieste di accesso ai blocchi di un disco mediante le primitive *ReadB* e *WriteB*[8] facenti riferimento al numero di *cilindro*, essenziale per posizionare correttamente la testina di *lettura/scrittura*.

In base alla natura dinamica degli accessi al disco si supponga di servire la coda delle richieste di spostamento della testina muovendola nella stessa direzione da un'estremo all'altro finché non raggiunge quest'ultimo; dopodicché la si riposiziona sul primo estremo e si ricomincia[9].

L'implementazione delle procedure di accesso ai cilindri del disco utilizza lo schema di sincronizzazione a prioritá appena visto.

```
type DiskHead = monitor;

   var Count : integer;
         Head : integer;
         Busy : boolean;
         Move : condition;

   procedure entry MoveTo( Cyl : Cylinder );
   begin
      if Busy then
         if Head <= Cyl then
            Move.wait( Cyl + Count ) else
            Move.wait( Cyl + Count + N );
         Busy := true;
         if Cyl < Head then
            Count := Count + N;
         Head := Cyl
   end;

   procedure entry Detach;
   begin
      Busy := false;
      Move.signal
   end;

begin
   Count := 0;
   Busy := false;
   Head := 0
end;
```

dove *Cylinder* enumera tutti i cilindri del disco, da 0 a N - *1*[10].

L'introduzione della variabile ausiliaria *Count* serve da dividere le richieste in due gruppi, quelle al di sotto e al di sopra della posizione corrente della testina, in modo tale che le prime siano trattate come un posizionamento della testina incrementato di N.

[8]che leggono, rispettivamente, scrivono un blocco su disco

[9]durante la fase di riposizionamento non viene servita alcuna richiesta

[10]avendo indicato con N il numero di cilindri del disco

5.3 MONITOR DI BRINCH HANSEN

Il meccanismo di sincronizzazione realizzato nel Concurrent Pascal é leggermente più semplice rispetto alle variabili *condizione* dei monitor di Hoare. Allo scopo vengono introdotte variabili di tipo *queue* la cui manipolazione può effettuarsi solamente con le operazioni **delay**[11] e **continue**[12].

La differenza sostanziale con le variabili *condition* di Hoare è che una variabile di tipo *queue* può avere al piú un solo processo per volta sospeso su di essa. Con questa definizione si puó facilmente realizzare la schedulazione di medio termine introducendo un vettore di *queue* ed eseguire l'operazione continue su quell'elemento che contiene il prossimi processo da risvegliare[13].

Rispetto a *signal*, l'operazione **continue**

5.3.1 Implementazione

Nel nucleo le variabili di monitor sono rappresentate da una struttura dati[14] che permette ad un solo processo per volta di accedere al monitor. Un *Cancello* é individuato da un valore booleano che determina se questo é *aperto* e da una coda di processi in attesa di entrare se questo é *chiuso*. All'inizio e alla fine delle procedure di monitor un processo esegue, rispettivamente, le operazioni indicate nel seguito.

```
type  Gate = record
                 Opened : boolean;
                 Event : Queue
             end;

procedure Enter( var G : Gate );
var Pid : Process;
begin
   with G do
      if Opened then
         Opened := false else
         begin
            Preempted( Pid );
            Insert( Event, Pid )
         end
end;

procedure Leave( var G : Gate );
var Pid : Process;
begin
   with G do
      if IsEmpty( Event ) then
         Opened := true else
         begin
```

[11] analoga a wait

[12] analoga a signal

[13] Si noti che gli elementi del vettore possono essere ordinati per valori decrescenti di prioritá per cui la prima variabile *queue* che ha un processo in attesa é quella su cui verrá eseguita l'operazione continue

[14] chiamata *Gate*, o *Cancello*

```
                Remove( Event, Pid )
                Activate( Pid );
            end
    end;
```

Nella prima, se il cancello é aperto, il processo entra e lo chiude; altrimenti, il processo viene bloccato in attesa che questo venga aperto. Nella seconda, se nessuno é in attesa sul cancello, questo viene lasciato aperto; altrimenti, viene selezionato un processo in attesa di riprendere l'esecuzione.

Le operazioni che realizzano il meccanismo di sincronizzazione, invece, sono definite dal frammento di codice riportato di seguito

```
procedure Delay( var Gt : Gate; var Pid : Process );
begin
    Preempted( Pid );
    Leave( Gt )
end;

procedure Continue( var Gt : Gate; var Pid : Process );
begin
    if Pid = nil then
    Leave( Gt ) else
        begin
            Activate( Pid );
            Pid := nil
        end
end;
```

e si applicano alle variabili di tipo *Queue*, definite all'interno del monitor. Si ricordi che la procedura *Delay* blocca l'esecuzione del processo corrente e lo introduce in una coda di processi contenente un solo processo. Il monitor puó ora passare sotto il controllo di un altro processo. L'operazione *Continue* forza il processo corrente ad abbandonare il monitor e ristabilisce l'esecuzione di un processo che era in attesa sulla corrispondente coda singola.

5.4 MONITOR DI WIRTH

Il Modula é un linguaggio di programmazione sviluppato per applicazioni su piccoli sistemi di elaborazione dedicati ed é fortemente basato sul Pascal e comprende processi, *moduli di interfaccia*, che sono molto simili ai monitor, e *moduli dispositivo*, che sono speciali moduli di interfaccia per la programmazione dei *driver*.

Il fatto che le variabili dichiarate nei moduli di interfaccia possano essere esportate fa si che queste siano referenziabili, anche se non modificabili, al di fuori dello *scopo* del modulo d'interfaccia che le definisce. Questa proprietá consente l'accesso concorrente a tali variabili che, peró, potrebbero condurre a problemi di interferenza a meno che queste vengano esplicitamente eliminate in fase di scrittura del programma.

Ciò premesso, si tenga presente che nell'implementazione precedente, sebbene una variabile *Signal* é realizzata come un lista di processi, questa puó interpretarsi come una condizione su cui quei processi possono fermarsi cossicché le operazioni **wait** e **send** diventano parzialmente equivalenti alle operazioni su variabili *condition* nei monitor di Hoare.

5.4.1 Implementazione

Il supporto run-time del Modula é piccolo ed efficiente e, come si é giá accennato nel Capitolo 2, non realizza la multiprogrammazione per quanti di tempo come accade nel caso del Concurrent Pascal. Al suo posto, vengono due fornite due operazioni definite nel nucleo, wait e send, il cui effetto é quello di assegnare il processore ad un altro processo o, come si é visto, *coroutine*.

Piú in dettaglio , tali operazioni sono implementate come riportato nel seguito[15].

```
procedure Send( var Pid : Signal );
var Curr : Signal;
begin
    if ( Pid <> nil ) then
        begin
            Curr := Ring;
            Ring := Pid;
            with Ring↑ do
                begin
                    Pid := Event;
                    Next := Curr↑.Next;
                    Ready := true;
                    Event := nil
                end;
            Transfer( Curr↑.Proc, Ring↑.Proc )
        end
end;

procedure Wait( var Pid : Signal );
begin
    if ( Pid = nil ) then
        Pid := Ring else
        Enqueue( Ring, Pid );
    Ring↑.Event := nil;
    Ring↑.Ready := false;
    Select( Curr, Ring );
    Transfer( Curr↑.Proc, Ring↑.Proc )
end;

function Awaited( Pid : Signal ) : boolean;
begin
    Awaited := Pid <> nil
end;

procedure InitSignal( var Pid : Signal );
```

[15]Si faccia riferimento alla parte del Cap. 2 relativa all'implementazione delle coroutine

begin
Pid := **nil**
end;

Si tenga presente che wait inserisce il processo corrente nella coda associata al verificarsi di un certo evento[16], passata come parametro della procedura; dopodicché scorrendo l'anello dei processi cerca il prossimo pronto ad eseguire, trasferendo ad esso il controllo.

L'operazione send attiva, se la sequenza di descrittori passata come parametro é non nulla, il processo associato al primo descrittore, trasferendone il controllo alla coroutine corrispondente. Le ultime due servono, rispettivamente, a verificare se la sequenza di descrittori di processo associata al parametro passato é vuota e ad inizializzare una sequenza di descrittori di processo.

5.5 ATTESA CONDIZIONALE DI HOARE

Per ritardare l'esecuzione di un processo all'interno del monitor viene introdotta l'istruzione di attesa condizionale

$$\textbf{wait}(B)$$

dove B è un'espressione booleana in cui compaiono variabili permanenti, o, locali, al monitor.

L'esecuzione di wait(B) ritarda il processo che l'ha invocata fino a quando B non diventa vera; non é richiesta, pertanto, alcuna operazione *signal* che riattivi i processi che sono stati ritardati in seguito ad un'istruzione di attesa condizionale.

Questo tipo di sincronizzazione é molto dispendioso in quanto

- é necessario valutare B ogni volta che un processo abbandona il monitor o si blocca in corrispondenza di un'attesa condizionale;
- la presenza di variabili locali alla condizione B, dipendenti da ciascun processo, puó richiedere uno scambio di contesto ad ogni valutazione.

5.6 MONITOR DI MITCHELL

E' noto che il meccanismo di sincronizzazione di Mitchell nei monitor si ottiene sostituendo il costrutto wait/signal dei Monitor di Hoare con i costrutti **wait/notify**, dove *wait* si comporta analogamente all'omonima operazione introdotta da Hoare. Per quanto riguarda, invece, *notify* questa non provoca mai l'attesa del processo che l'ha invocata.

5.6.1 Implementazione del Monitor di Mitchell

Si mostri quali semafori e variabili é necessario introdurre[17] per realizzare il monitor di Mitchell[18]. A tal fine si proceda secondo lo schema indicato di seguito.

- Si introduca una variabile *Count* e un semaforo *Turn* per ciascuna variabile di tipo condition.

[16] e, pertanto, identificata da una opportuna variabile di tipo *Signal*
[17] 16 febbraio 90
[18] si supponga, per semplicitá di avere soltanto una variabile di tipo **condition**

- Si definiscano i semafori *Mutex* e *Urgent* per controllare l'accesso al monitor da parte dei processi che ne fanno richiesta. Si introduca, eventualmente, qualche altra variabile che tenga conto delle prioritá di accesso.
- Si implementino le operazioni *cond* **.wait** e *cond* **.signal** tenendo presente che il processo che esegue quest'ultima, pur segnalando ad un processo giá in attesa che puó continuare, non rilascia il controllo del monitor se non quando ha terminata la relativa procedura oppure va in attesa su un'altra condizione.

Per quanto riguarda la struttura di ciascuna procedura di monitor visibile dall'esterno é sufficiente indicare in che modo viene realizzata la mutua esclusione.

```
type Condition = record
          Jobs : integer;
          Turn : semaphore
      end;

var   Mutex : semaphore;
          Urgent : semaphore;
          Count : integer;

procedure Entry( procedure Proc );
begin
   P( Mutex );
   Proc;
   if Count = 0 then
      V( Mutex ) else
      V( Urgent )
end;

procedure Wait( var Cond : Condition );
begin
   with Cond do
      begin
         Jobs := Jobs + 1;
         if ( Count > 1 ) then
            V( Urgent ) else
            V( Mutex );
         P( Turn )
      end;
   P( Urgent );
   Count := Count - 1
end;

procedure Notify( var Cond : Condition );
begin
   with Cond do
      if Jobs > 0 then
```

```
            begin
                Count := Count + 1;
                Jobs := Jobs - 1;
                V( Turn )
            end
    end;
```

5.6.2 Realizzazione dello Scambio Messaggi mediante Monitor

Utilizzando il costrutto di monitor con il meccanismo di sincronizzazione secondo Mitchell si realizzi uno schema di comunicazione fra processi basato sullo scambio messaggi, eventualmente potendo disporre di porte. Si definisca, a questo proposito, la porta come un monitor in modo che ciascuna sua istanza individui una variabile che rappresenta una porta particolare.

Introducendo le opportune variabili e le relative procedure si implementi tale monitor seguendo lo schema riportato di seguito.

```
    type Port = monitor;
    { dichiarazioni di variabili globali }
        procedure entry Send( Msg : Message );
        procedure entry Release( var Msg : Message );
    { inizializzazione delle variabili globali }
    end
```

Si tenga presente che lo schema adottato da ciascun Cliente e dal Server per accedere alla porta é il seguente

```
    Cliente             Server
    ....                ....
    T.Send( Msg )       T.Receive( Msg )
    ....                ....
```

dove T é l'opportuna variabile di tipo Monitor che rappresenta la porta cui i Clienti e il Server fanno riferimento. Si noti, infine, come tale struttura presupponga la definizione del tipo *Message* cui la variabile *Msg* fa riferimento.

5.7 MONITOR DI KESSEL

5.7.1 Il Problema dei Lettori/Scrittori

Un certo numero **n** di processi[19] cooperano su un insieme comune di dati. I processi possono essere classificati in due categorie, chiamate Lettori e Scrittori, i quali utilizzano i dati secondo lo schema seguente.

- Uno scrittore deve avere l'accesso esclusivo sui dati, mentre i lettori possono condividerne l'accesso.
- Non appena uno scrittore é pronto a scrivere, nessun'altro lettore puó ottenere il permesso ad usare i dati.

[19] 16 ottobre 88

- Uno scrittore deve attendere che tutti i precedenti scrittori e tutti i lettori rilascino i dati prima di scrivere su di essi.

Si supponga di sostituire il costrutto wait/signal dei Monitor col l'unico costrutto **await** B, dove B é un'espressione booleana qualunque che provoca l'attesa del processo fino a quando B non diventa vera[20].

- Si definiscano le opportune variabili e le relative procedure di monitor per implementare il problema dei Lettori/Scrittori, seguendo lo schema riportato di seguito.

```
type ReaderWriter = monitor;
{ dichiarazioni di variabili locali }
    procedure entry OpenRead;
    procedure entry CloseRead;
    procedure entry OpenWrite;
    procedure entry CloseWrite;
{ inizializzazione delle variabili }
end;
```

Si tenga presente che lo schema adottato da ciascun Lettore e ciascun Scrittore per accedere alle variabili comuni é il seguente

Lettore	Scrittore
....	....
T.OpenRead	T.OpenWrite
ReadDatum	WriteDatum
T.CloseRead	T.CloseWrite
....	....

dove T é l'opportuna variabile di tipo Monitor che rappresenta i dati comuni su cui i Lettori/Scrittori cooperano.

- Si spieghi perché il costrutto **await** B, precedentemente definito, non puó essere implementato efficientemente in generale. Si consideri a questo proposito in che modo la valutazione di B dipende dalle variabili che ivi compaiono.

- Quali sono le restrizioni che devono essere introdotte sulla forma di B affinché l'istruzione **await** sia implementabile efficientemente?

- Nel caso che siano verificate le condizioni di cui si parla al punto precedente, sia dia una possibile implementazione del costrutto **await** in termini di semafori.

```
type ReaderWriter = monitor;

  var Readers : integer;
        Writers : integer;
        Busy : boolean;
        Reading : condition ( Writers = 0 );
```

[20] meccanismo di sincronizzazione secondo Kessel

```
        Writing : condition ( Readers = 0 ) and not Busy;

    procedure entry OpenRead;
    begin
        await Reading;
        Readers := Readers + 1
    end;

    procedure entry CloseRead;
    begin
        Readers := Readers - 1
    end;

    procedure entry OpenWrite;
    begin
        Writers := Writers + 1;
        await Writing;
        Busy := true
    end;

    procedure entry CloseWrite;
    begin
        Busy := false;
        Writers := Writers - 1
    end;

begin
    Readers := 0;
    Writers := 0;
    Busy := false
end;
```

- L'inefficienza é dovuta principalmente al fatto che B puó dipendere da dati privati dei processi chiamanti, cioé da dati locali della procedura di monitor che viene eseguita in quanto questi possono dipendere dai parametri che il processo passa alla procedura. Questo fatto implica che ogni qualvolta un processo rilascia il controllo del monitor l'espressione B deve essere rivalutata da ciascun processo in attesa su quella condizione. Inoltre, é richiesto uno scambio di contesto (dei processi) ad ogni rivalutazione, dal momento che ciascun processo puó rivalutare la propria istanza dell'espressione B.

- Per superare le difficoltá citate al punto precedente é necessario imporre che B non dipenda da dati locali delle procedure di monitor, cioé dai dati privati del processo chiamante. A tal scopo é sufficiente richiedere che ciascuna *condizione* sia dichiarata all'inizio de monitor nel modo seguente

  $$< Identifier >: \textbf{condition} < BooleanExpression >$$

  Ogni variabile di tipo condizione puó essere utilizzata dalle procedure del monitor per mezzo dell'istruzione

  $$\textbf{await} < ConditionIdentifier >$$

senza nessuna restrizione sul loro numero in ciascuna procedura di monitor. L'esecuzione di una istruzione **await** invoca l'esecuzione della corrispondente espressione *booleana* la quale, se risulta verificata determina la continuazione del processo; altrimenti, il processo viene sospeso, cioé inserito nella coda di attesa dei processi che attendono su quella condizione.

```
var Mutex : semaphore;
      Cond : semaphore;
      Cond1 : semaphore;
      Count : integer;
      Count1 : integer;

   procedure entry <Proc>( <Parms> );
   begin
      <Statements>;
      await B;
      <Statements>;
   end;

   procedure entry <Proc>( <Parms> );
   begin
      P( Mutex );
      <Statements>;
      if not B then
         begin
            Count := Count + 1;
            if ( Count1 > 0 ) and B1
               then V( Cond1 )
               else V( Mutex );
            P( Cond );
            Count := Count - 1
         end;
      <Statements>;
      if ( Count1 > 0 ) and B1
         then V( Cond1 )
         else V( Mutex )
   end;
```

5.7.2 Attraversamento di un Ponte a Senso Unico Alternato

Si consideri il problema dell'attraversamento di un fiume con un ponte che permette il passaggio degli autoveicoli a senso unico alternato. Si indichino con *North* e *South* le due sezioni di strada, a doppio senso di marcia, cui il ponte é connesso. Si supponga, inoltre, che la direzione del traffico attraverso il ponte cambi quando al piú N macchine[21] hanno attraversato il ponte in una direzione mentre una o piú macchine stanno aspettando di attraversare il ponte nella direzione opposta.

Utilizzando il costrutto di *monitor* con il meccanismo di sincronizzazione secondo Kessel, si arbitri l'accesso al ponte, definendo opportunamente una risorsa di tipo *Bridge* in modo tale che le autovetture,

[21] con $N > 0$

senza privilegio per la direzione di provenienza[22] possano accedere e attraversare il ponte senza conflitti di direzione. In particolare, supponendo assegnata la seguente struttura per il monitor

```
type CrossBridge = monitor;
{ definizioni di tipo e dichiarazioni di variabili permanenti }
   procedure entry FromSouth;
   procedure entry ToNorth;
   procedure entry FromNorth;
   procedure entry ToSouth;
{ inizializzazione delle variabili permanenti }
end;
```

la si completi

- definendo adeguatamente il tipo *Direction* per le direzioni *South* e *North*[23] in modo tale che tenga conto delle macchine in *Attesa*[24], in *Attraversamento*[25] e quelle *davanti* giá in movimento[26];
- dichiarando le opportune variabili di tipo **condition** come é richiesto dal costrutto **await** di Kessel;
- realizzando le procedure *FromSouth* e *FromNorth* invocate da ogni autovettura che intende impegnare l'accesso al ponte;
- realizzando le procedure *ToNorth* e *ToSouth* che permettono a ciascuna autovettura di rilasciare il ponte;
- inizializzazando opportunamente le variabili permanenti del monitor.

```
type CrossBridge = monitor;
   type Direction = record
         Waiting : integer;
         Crossing : integer;
         Ahead : integer;
      end;

   var South : Direction;
         North : Direction;
         Southern : condition ( North.Crossing = 0 ) and
                      ( South.Ahead ¡ N );
         Northern : condition ( South.Crossing = 0 ) and
                      ( North.Ahead ¡ N );
   procedure entry FromSouth;
   begin
      with South do
```

[22]e, quindi, fornendo una soluzione simmetrica per i due flussi di traffico
[23]che devono essere realizzate per mezzo di opportune variabili
[24]*Waiting*
[25]*Crossing*
[26]*Ahead*

```
            begin
                Waiting := Waiting + 1;
                await Southern;
                Waiting := Waiting - 1;
                Crossing := Crossing + 1;
                if North.Waiting ¿ 0 then
                Ahead := Ahead + 1
            end
    end;

    procedure entry ToNorth;
    begin
        with South do
            begin
                Crossing := Crossing - 1;
                if Crossing = 0 then
                North.Ahead := 0
            end
    end;

    procedure entry FromNorth;
    begin
        with North do
            begin
                Waiting := Waiting + 1;
                await Northern;
                Waiting := Waiting - 1;
                Crossing := Crossing + 1;
                if South.Waiting ¿ 0 then
                Ahead := Ahead + 1
            end
    end;

    procedure entry ToSouth;
    begin
        with North do
            begin
                Crossing := Crossing - 1;
                if Crossing = 0 then
                South.Ahead := 0
            end
    end;

begin
    with South do
        begin
            Waiting := 0;
```

```
            Crossing := 0;
            Ahead := 0
        end;
begin
    with North do
        begin
            Waiting := 0;
            Crossing := 0;
            Ahead := 0
        end;
end;
```