Capitolo 3

SCAMBIO MESSAGGI

Si é visto, nel capitolo precedente, che il termine *scambio messaggi* identifica una comunicazione fra processi che si basa su un modello computazionale di tipo data-driven il quale, permettendo il trasferimento[1] di valori di variabili (riferite da un processo) a variabili riferite da un altro processo, introduce un **meccanismo implicito di sincronizzazione** nel trasporto di dati fra processi.

3.1 Canali di Comunicazione

In questo modo rimane automaticamente associato un *meccanismo di sincronizzazione* ad un *meccanismo di trasporto dei dati* secondo il quale i processi *inviano* e *ricevono* messaggi invece di scrivere o leggere variabili comuni. Pertanto,

lo scambio messaggi realizza

- una **comunicazione** perché un processo, al momento di ricevere un messaggio, ottiene i valori di cui abbisogna da qualche altro processo;
- una **sincronizzazione** perché un messaggio puó essere ricevuto solamente dopo che sia stato inviato (il che impone l'ordine con cui questi eventi devono verificarsi)

L'istruzione generale per inviare messaggi é la seguente

$$\textbf{send} \quad < expr - list > \quad \textbf{to} \quad < dest - designator >$$

dove i valori delle espressioni, all'istante in cui viene eseguita l'istruzione **send**[2], in ¡expr-list¿ sono il contenuto del messaggio. Il designatore di destinazione permette di controllare il punto in cui il messaggio deve pervenire, ossia le istruzioni che devono ricevere i valori trasmessi.

L'istruzione generale per ricevere un messaggio é la seguente

$$\textbf{receive} \quad < var - list > \quad \textbf{from} \quad < source - designator >$$

dove ¡var-list¿ é la lista delle variabili cui vengono assegnati i valori contenuti nel messaggio al momento cui questo viene ricevuto[3]. Il designatore di provenienza permette di controllare il punto di partenza del messaggio e quindi le istruzioni coinvolte nell'invio del messaggio.

[1] assegnazione, per cui si parla anche di *assegnazione remota*

[2] Accanto alla forma imperativa indicata nel testo, é possibile anche la forma funzionale **send**($< destination >$, $< expr - list >$)

[3] Accanto alla forma imperitiva indicata nel testo, é possibile anche la forma funzionale **receive**($< source >$, $< var - list >$)

La ricezione del messaggio provoca, innanzitutto,

- l'*assegnazione* dei valori contenuti nel messaggio alla lista delle variabili che compaiono nell'istruzione **receive**; successivamente,
- la *distruzione* del messaggio.

La scelta delle primitive di scambio messaggi é definita dalla forma e dal significato di queste due istruzioni generali; in particolare, la scelta é legata al modo di

- **specificare** i designatori di provenienza e di destinazione,
- **sincronizzare** la comunicazione.

La coppia di designatori ¡*provenienza, destinazione*¿ definisce un **canale di comunicazione** il cui nome é identificato univocamente da tale coppia. In generale, lo schema scelto per identificare il canale caratterizza il tipo di comunicazione.

Nel caso piú semplice si utilizzano come designatori di provenienza e di destinazione i nomi dei processi coinvolti nella comunicazione. In questo caso, allora, si parla di **denominazione diretta**[4] del canale. La denominazione diretta dei canali é facile da utilizzare ed implementare; permette, inoltre, che un processo controlli direttamente gli istanti di tempo in cui riceve messaggi dagli altri processi.

I due paradigmi fondamentali su cui é possibile modellare l'interazione fra processi concorrenti, noti in letteratura col nome di *pipeline*[5] e *relazione client/server*, dipendono strettamente dal modo con cui viene rappresentato il canale di comunicazione.

Si definisce **pipeline** *un insieme ordinato di processi concorrenti in cui i dati di uscita di ciascun processo sono utilizzati come dati di ingresso del successivo.*

In questo caso le informazioni fluiscono fra i processi analogamente al modo con cui un liquido fluisce entro una tubatura da cui segue immediatamente che i *pipeline* sono implementabili facilmente mediante canali di comunicazione a *denominazione diretta*.

Si dice che un insieme di processi é in relazione **client/server** *se alcuni processi, detti* server*, rendono un* servizio *a qualche processo* cliente *che ne ha fatto richiesta con un opportuno messaggio. Il* server *riceve la richiesta, esegue il servizio e, se necessario, restituisce un messaggio di servizio eseguito al cliente.*

In questa situazione, invece, é immediato verificare che non é possibile utilizzare la denominazione diretta, in quanto

- l'istruzione **receive**, che compare in ciascun server, deve permettere di ricevere messaggi da qualunque cliente;
- l'istruzione **send**, eseguita da ciascun cliente, deve permettere di far riferimento a qualunque server, quando esiste un insieme di server dello stesso tipo che realizzano le medesime funzioni.

La generalizzazione dei canali a denominazione diretta che permette di realizzare schemi di comunicazione di tipo client/server é ottenuta per mezzo di canali di comunicazione che utilizzano *nomi globali*[6].

Si dice che uno schema di comunicazione é realizzato per mezzo di **nomi globali** *se tali nomi possono comparire come designatori di destinazione e di provenienza, rispettivamente, in istruzioni* **send** *e* **receive**.

[4]in letteratura si trova il termine inglese *direct naming*
[5]termine inglese avente il significato di *tubatura, conduttura*
[6]oppure *cassette postali*, dal termine inglese **mailbox**

Cosí, i messaggi inviati ad una prefissata cassetta postale possono essere ricevuti da qualunque processo che esegue una **receive** che faccia riferimento a quella cassetta postale.

Seguendo questo schema, l'interazione *client/server* é realizzata in modo tale che

- i clienti inviino le richieste di servizio ad una prefissata *cassetta postale*,
- i server ricevano tali richieste dalla medesima cassetta postale.

Sfortunatamente la realizzazione di cassette postali puó risultare oltremodo complessa e inefficiente se non si ha a disposizione una una rete di comunicazione specializzata e questo dipende dal fatto che la presenza del messaggio, al momento in cui viene spedito, deve essere notificata ad ogni istruzione **receive** che puó essere eseguita su quella cassetta postale. Dopodiché, appena il messaggio é stato ricevuto da qualche istruzione **receive**, bisogna notificare immediatamente a tutte le altre istruzioni, che possono ricevere messaggi dalla stessa cassetta postale, che il messaggio non é ulteriormente disponibile.

Una cassetta postale prende il nome di **porta** *quando questa appare come designatore di provenienza in istruzioni* **receive** *di un solo processo.*

Le porte sono facili da implementare poiché tutte le istruzioni **receive** che designano una porta possono apparire soltanto in un unico processo. Questa restrizione permette un'immediata realizzazione dello schema a server singolo, mentre non é in grado di fornire una soluzione diretta allo schema generale in cui sono presenti piú server.

Puó essere utile, a questo punto, entrare in maggior dettaglio sulla forma che possono assumere i *server*. Fin d'ora, comunque, appare chiaro il ruolo che questi processi assumono nel caratterizzare la struttura e le caratteristiche dell'ambiente in cui i processi sequenziali *utente* andranno a collocarsi per portare a termine i loro compiti. I processi *server* sono strutturalmente simili alle unitá di elaborazione che controllano l'accesso ai dispositivi periferici degli elaboratori elettronici e che, per questo, prendono il nome di **device controller**. In generale, si puó dire che la forma di un server puó rappresentarsi con la seguente procedura PASCAL.

```
procedure Server;
{ dichiarazione di variabili locali }
begin
    Init;
    repeat
        receive( Service, Request, PId );
        CarryOut( Request, Answer );
        send( PId, Answer )
    until false
end;
```

La porta *Service* determina il punto d'accesso del server[7] dove i clienti depositano le loro richieste; la procedura *CarryOut* rappresenta, invece, la funzione cui quel particolare server é preposto mentre *PId* identifica il processo che ha fatto richiesta di servizio a quel particolare server.

In vista di applicazioni successive é conveniente trattare esplicitamente alcuni casi specifici, in cui il significato dei designatori di destinazione e di provenienza assume particolare interesse. Si supponga, innanzittutto, di utilizzare le cassette postali per memorizzare il verificarsi, o meno, di un certo evento.

[7] corrispondente all'appropriato **bus**, nel caso di *device controller*

In questo caso le operazioni di scambio messaggi si riducono ad un semplice meccanismo di notifica che un certo *evento* si é verificato.

Si dice che una cassetta postale definisce un **evento** quando l'informazione scambiata ha la proprietá di caratterizzare l'istante temporale in cui tale evento si é verificato. Inoltre, le primitive che realizzano lo scambio messaggi sono

$$\begin{aligned} \mathbf{cause}(< \mathit{Event} >) &\doteq \mathbf{send}(< \mathit{Event} >, \mathbf{true}) \\ \mathbf{await}(< \mathit{Event} >) &\doteq \mathbf{receive}(< \mathit{Event} >, \mathit{Succeded}) \end{aligned}$$

dove *Succeded* é una variabile booleana che permette di identificare quando l'evento si viene a verificare.

Come apparirá chiaro nel seguito, l'evento che notifica ad un processo di continuare la propria esecuzione assume un significato fondamentale nell'ambito della programmazione concorrente. Indicheremo con **continue** l'evento che permette ad un processo di continuare l'esecuzione su un prefissato processore virtuale. In generale, dunque, consideremo un evento di questo tipo per ciascun processore virtuale a disposizione.

Se i designatori di provenienza e destinazione sono fissati durante la fase di compilazione, si dice che la *denominazione del canale* é **statica**; se, invece, tali designatori sono determinati durante l'esecuzione, si parla di **denominazione dinamica del canale**.

Benché la denominazione statica dei canali sia ampiamente diffusa, essa presenta alcuni inconvenienti. Innanzitutto, viene preclusa la possibilitá che i programmi possano comunicare su canali che non siano stati definiti prima della compilazione limitandoli ad esistere in ambienti di esecuzione immutabili. Inoltre, se un programma necessita di accedere ad un canale, allora vi puó accedere *permanentemente*[8].

3.2 Sincronizzazione

Affinché sia possibile realizzare la denominazione dinamica dei canali é necessario arricchire lo schema sottostante a denominazione statica con variabili che possano contenere i designatori di provenienza e destinazione. Queste variabili possono vedersi come **capacitá**[9] di comunicazione.

Un'altra proprietá fondamentale delle istruzioni di scambio messaggi riguarda il fatto che la loro esecuzione puó provocare un ritardo. Si dice, allora, che l'istruzione é **non bloccante** se la sua esecuzione non ritarda mai il processo che l'ha invocata; altrimenti, si dice che l'istruzione é **bloccante**.

In alcuni schemi di scambio messaggi, essenzialmente quelle implementazioni che possono fare affidamento su un supporto comune di memoria, i messaggi possono risiedere temporaneamente su **memorie tampone**[10] la cui funzione é quella di assorbire le variazioni di velocitá fra i processi. Si consideri a questo proposito, il seguento frammento di programma PASCAL che definisce un appropriata struttura di supporto per i dati in transito fra processi.

```
type    Index  = 0 .. N - 1;
        Size   = 0 .. N;
        Buffer = array[ Index ] of Message;
      Mailbox  = record
                   B  :  Buffer;
                   l  :  Index;
```

[8] Si noti che in molte applicazioni, come possono essere i *file-system* é preferibile poter allocare dinamicamente canali di comunicazione con le risorse, in questo caso *file*

[9] in letteratura si trova il termine inglese *capability*

[10] in letteratura si trova il termine inglese *buffer*

```
    c : Index;
    s : Size
end;
```

Avendo indicato con *Message* la forma del generico messaggio, la cassetta postale contiene, oltre alla lista dei messaggi pervenuti non ancora smistati, il loro numero **s**, il riferimento alla prima posizione disponibile **l** per il prossimo messaggio in arrivo, il riferimento al primo messaggio da recapitare **c** non appena qualche processo ne faccia richiesta. Naturalmente, poiché la memoria tampone ha capacitá limitata, é necessario imporre alcune *restrizioni* sulla comunicazione; in particolare,

- i processi che inviano messaggi non possono superare la capacitá della memoria tampone,
- i destinatari della comunicazione non possono ricevere i messaggi prima che questi siano pervenuti,
- nessuno dei processi coinvolti puó interferire nella comunicazione.

Caratterizzando, pertanto, gli eventi *messaggio disponibile*, *spazio disponibile* e *non interferenza*, rispettivamente, con **s** > 0, **s** $<$ N, *Writing* e *Reading*, le operazioni *Send* e *Receive* possono essere realizzate mediante le due procedure PASCAL riportate nel seguito. Si osservi, a questo proposito, che gli eventi sopracitati controllano la comunicazione mediante la seguente **regola di sincronizzazione**:

Se il processo che tenta di inviare un messaggio trova la memoria tampone piena*, allora questo viene ritardato fino a quando un processo destinatario non estrae un messaggio; d'altra parte, se il destinatario tenta di prelevare un messaggio quando la memoria tampone é* vuota*, allora questo viene ritardato in attesa che qualche processo ne depositi uno.*

Le operazioni di *deposito* e *prelievo* messaggio ipotizzano una organizzazione **circolare** della memoria tampone cosicché, ogni volta che é stata scritta la locazione N si attende la disponibilitá della locazione *0*.

```
procedure Send( var Mbx : Mailbox; Msg : Message );
begin
   with Mbx do
      begin
         await( s < N );
         await( Writing );
         B[ c ] := Msg;
         c := ( c + 1 ) mod N;
         s := s + 1;
         cause( Writing );
         cause( s > 0 )
      end
end;
procedure Receive( var Mbx : Mailbox; var Msg : Message );
begin
   with Mbx do
      begin
         await( s > 0 );
         await( Reading );
         Msg := B[ l ];
```

```
            l := ( l + 1 ) mod N;
            s := s - 1;
            cause( Reading );
            cause( s < N )
        end
    end;
```

Quindi, se al momento dell'esecuzione di una istruzione **send**, la memoria tampone risulta essere piena, tale istruzione ritarda il processo che l'ha invocata fino a quando nella memoria tampone non c'é spazio per il messaggio. Naturalmente, é possibile che l'istruzione **send** restituisca un codice che notifichi, a seguito di indisponibilitá di spazio nella memoria tampone, che il messaggio non puó essere spedito.

In modo del tutto analogo se, al momento dell'esecuzione di una istruzione **receive**, non é disponibile alcun messaggio la cui provenienza é quella indicata, il processo che l'ha invocata viene ritardato oppure riceve in risposta un codice che notifica la non disponibilitá di messaggi provocando, contemporaneamente, la terminazione dell'istruzione.

Nell'ipotesi che il sistema disponga di memorie tampone di capacitá illimitata, allora il processo che invia messaggi, eseguendo una istruzione **send**, non viene mai ritardato. Questa forma di comunicazione prende il nome di **scambio messaggi asincrono**[11]. Questo tipo di scambio messaggi permette al processo mittente di proseguire arbitrariamente l'esecuzione, al di lá dello stato corrente del destinatario il quale, al momento della ricezione puó osservare uno stato del mittente che puó essere molto diverso dal suo stato attuale.

Vedremo, nel prossimo capitolo, come sia possibile implementare una forma di scambio messaggi di questo tipo, semplicemente, considerando *messaggi vuoti*. In questo caso non interessa tanto il contenuto del messaggio, che é nullo per le ipotesi fatte, ma il loro numero. Le procedure PASCAL, precedentemente introdotte, che implementano le operazioni **send** e **receive**, si semplificano nel codice seguente.

```
procedure Send( var s : integer );
begin
    await( Continue );
    s := s + 1;
    cause( s > 0 )
end;
procedure Receive( var s : integer );
begin
    await( s > 0 );
    s := s - 1;
    cause( Continue )
end;
```

Si noti come la cassetta postale sia stata sostituita da una variabile intera, mentre la memoria tampone con relativi puntatori e l'evento spazio disponibile diventano inessenziali. Variabili di questo

[11] oppure **send no-wait**, in virtú del fatto che non si attende mai l'esito del trasferimento del messaggio dopo che questo é stato inviato

tipo giocano un ruolo fondamentale nei problemi di sincronizzazione per cui Dijkstra ha suggerito di chiamarle **semafori**.

3.3 Comunicazione fra Processi

L'ambiente di comunicazione fra processi basato sullo scambio messaggi si puó ottenere utilizzando il meccanismo delle **socket**, il cui scopo é quello di introdurre punti terminali di comunicazione ai quali viene *attribuito*[12] un nome. A ciascuna socket, che esiste all'interno di un **dominio di comunicazione**, viene associato un **tipo** ed uno o piú processi.

Il dominio di comunicazione é un'astrazione che viene introdotta per raggruppare le proprietá comuni a tutti i processi che comunicano con un certo tipo di socket. Una di queste proprietá é lo schema utilizzato per denominare le socket. Normalmente le socket scambiano dati solamente con altre socket all'interno dello stesso dominio. Per poter scambiare dati oltre tali limiti é richiesto un processo di traduzione.

Le socket sono tipizzate secondo le proprietá di comunicazione visibili all'utente. Si presume che i processi comunichino solamente fra socket dello stesso tipo benché non vi sia alcuna preclusione alla comunicazione fra socket di tipo diverso purché il sottostante protocollo di comunicazione lo permetta. Normalmente sono disponibili tre tipi di socket che prendono il nome di *stream*, *datagram* e *sequenced packet*.

Nel primo caso, la socket **stream** fornisce un flusso di dati bidirezionale, affidabile, sequenziale e senza duplicazioni e senza limiti fissati di lunghezza di trasmissione. A parte la bidirezionalitá del flusso di dati, una coppia di socket stream interconnessa fornisce un'interfaccia molto simile a quella delle pipe.

Una socket di tipo **datagram** sostiene un flusso di dati bidirezionale per il quale non c'é alcuna garanzia che sia sequenzializzato, affidabile e senza duplicazioni. Questo significa che un processo che riceve messaggi da una socket di tipo datagram puó ritrovarsi con messaggi duplicati che possono apparire in un ordine diverso da quello in cui sosno stati spediti. Una caratteristica importante di questo tipo di socket é la non alterazione della delimitazione dei record costituenti il messaggio. Questa socket modella il comportamento di molte reti a commutazione di pacchetto come Ethernet.

Definito un dominio, vi si possono introdurre un certo numero di socket ed accedervi mediante le seguenti operazioni primitive:

- **procedure** *MakeSocket*(Env : Domain; **var** S : Socket), il cui effetto é quello di generare localmente al processo un descrittore[13] per far si che il processo possa accedere ad un canale di comunicazione;
- **procedure** *Bind*(S : Socket; Mbx : Mailbox), il cui effetto é quello di notificare all'esterno del processo il punto terminale del canale di comunicazione;
- **procedure** *Connect*(S : Socket; Mbx : Mailbox), il cui effetto é quello di creare la connessione fra il nome locale[14] ad un processo ed il corrispondente nome globale[15];
- **procedure** *Listen*(S : Socket; N : **integer**), il cui effetto é quello di verificare se c'é qualche messaggio in arrivo sul canale[16];

[12]viene utilizzato il termine **bound**
[13]di un tipo del tutto simile ai descrittori dei file aperti di un processo
[14]la propria socket
[15]legato alla socket di un altro processo
[16]questa operazione implica attesa sul canale

- **procedure** *Accept*(S : Socket; **var** P : Socket), il cui effetto é quello di ricevere il messaggio in arrivo sul canale;
- **procedure***Close* (S : Socket), il cui effetto é quello di chiudere il canale di comunicazione precedentemente aperto;
- **procedure** *GetB*(S : Socket; **var** Msg : Message), il cui effetto é quello di prelevare un messaggio dalla socket;
- **procedure** *PutB*(S : Socket; Msg : Message), il cui effetto é quello di introdurre un messaggio nella socket.

A questo punto si possono realizzare le operazioni di scambio messaggi precedentemente introdotte. In particolare, per quanto riguarda l'operazione **send**, tenendo presente che il cliente necessita della connessione solamente per il tempo necessario a spedire il messaggio, si ha

```
procedure Send( var Mbx : Mailbox; Msg : Message );
var S : Socket;
begin
   MakeSocket( DOMAIN, S );
   Connect( S, Mbx );
   PutB( S, Msg );
   Close( S )
end;
```

dove il nome della socket Mbx é, in parte, determinato dalla scelta di DOMAIN, il dominio prescelto all'interno del quale viene stabilita la comunicazione. Come si intuisce facilmente, ogni qualvolta il cliente necessita di inviare un messaggio alla mailbox *Mbx*, viene creata temporaneamente una socket di trasmissione entro la quale viene introdotto il messaggio; successivamente la socket viene distrutta.

Per quanto riguarda, invece, l'operazione **receive**, questa puó essere implementata nel modo seguente

```
procedure Receive( var S : Socket; var Msg : Message );
var P : Socket;
begin
   Accept( S, P );       { P = socket di comunicazione}
   if fork = 0 then      { S = socket di trasmisione}
   begin
      Close( S );
      GetB( S, Msg );
      Close( P )
   end
end;
```

nella quale si fa riferimento alla Mailbox con la corrispondente socket piuttosto che col suo nome globale. Questo é possibile purché si supponga che esiste un unico server che ascolta sulla socket, cioé, i nomi globali devono considerarsi identificativi di porte. Si deve aggiungere, pertanto, la procedura StartReceive che appare nel seguito

```
procedure StartReceive( Mbx : Mailbox; var S : Socket );
begin
    MakeSocket( DOMAIN, S );
    Bind( S, Mbx ); { S = socket di trasmissione}
    Listen( S, N )
end;
```

il cui compito é quello di attivare il punto di connessione sul quale il *server* esegue le accettazioni di messaggi, essendo N il loro numero massimo in attesa di ricezione.

3.4 Comunicazione Selettiva

All'altro estremo, quando non vi é alcuna bufferizzazione, l'esecuzione di una istruzione **send** provoca sempre un ritardo sul processo che l'ha invocata fino a quando la corrispondente istruzione **receive** non é stata eseguita. A questo punto il messaggio viene trasferito ed entrambi i processi procedono. Si dice, allora, che si é in presenza di uno **scambio messaggi sincrono**, il che significa che lo scambio del messaggio rappresenta un punto di sincronizzazione durante l'esecuzione dei processi mittente e ricevente. Perció, il messaggio ricevuto corrisponderá sempre allo stato corrente del mittente.

Fra questi due estremi si pone lo **scambio messaggi bufferizzato** in cui si é in presenza di buffer di dimensioni limitate che permettono al processo mittente di andare avanti rispetto allo stato del processo ricevente, ma non arbitrariamente avanti.

La forma piú comune dell'istruzione **receive** é quella bloccante perché, molto spesso, i processi riceventi non hanno null'altro da fare mentre attendono l'arrivo di qualche messaggio. Comunque, molti linguaggi e sistemi operativi mettono a disposizione dell'utente istruzioni **receive** non-bloccanti o qualche strumento ausiliario per verificare se l'esecuzione di una **receive** provocherá la sospensione del ricevente.

Questi costrutti, pur abilitando i processi a ricevere tutti i messaggi in arrivo, permettono di selezionarne[17] uno per l'effettiva elaborazione. In alcuni casi viene fornito un ulteriore controllo sui messaggi che possono essere ricevuti mediante un'istruzione del tipo

$$\textbf{receive}^{18} < var - list > \textbf{from} < source > \textbf{when} < bool - expr >$$

la quale consente soltanto la ricezione di quei messaggi che rendono l'espressione booleana ¡bool-expr¿ vera. Questa facilitazione permette che un processo possa pescare nel contenuto dei messaggi prima di riceverli. Benché la precedente non sia strettamente necessaria, le istruzioni **receive** condizionali danno luogo a soluzioni coincise in molti problemi di sincronizzazione.

Per quanto riguarda, invece, l'istruzione **receive**, questa puó essere bloccante oppure, non-bloccante. Nel primo caso, l'istruzione **receive** realizza implicitamente una *sincronizzazione* fra il mittente e il ricevente perché ritarda il proceso ricevente fino a quando il messaggio non viene effettivamente spedito. Nel secondo caso, quando si é in presenza di **receive** non-bloccante, la realizzazione della sincronizzazione é possibile purché venga impiegato un meccanismo di attesa attiva del tipo

```
repeat
    Done := receive( <source>, <var-list> )
```

[17] piú esattamente, a fare una schedulazione

[18] oppure, nella forma di pseudoprocedura **receive**(<source>, <var-list>, <bool-expr>)

until Done

dove si é fatto uso di un'istruzione **receive** modificata, in forma pseudo-funzionale, che restituisce valore **true** quando il messaggio é presente, **false** altrimenti.

Comunque, le istruzioni di scambio messaggi bloccanti possono dar luogo allo stesso effetto semantico di quelle non bloccanti utilizzando un meccanismo che prende il nome di **comunicazione selettiva** e che é basato sui *comandi con guardia* introdotti da Dijkstra. In quest'approccio si tiene conto del fatto che quando il controllo raggiunge un'istruzione di scambio messaggi, che potrebbe ritardare il processo che l'ha invocata, questo sarebbe in grado di avanzare solamente nel caso che le variabili richieste dalla comunicazione non siano immediatamente utilizzabili.

Questo é vero solamente nel caso che queste non appaiano immediatamente dopo l'istruzione di scambio messaggi, oppure, nel caso in cui vi siano altre istruzioni di questo tipo che, peró, non danno luogo a ritardi. Raggruppando le istruzioni di scambio messaggi in opportuni costrutti oppure anticipando l'esecuzione di quelle istruzioni che non hanno immediata necessitá di valori provenienti da transiti di dati, é il metodo che permette di *realizzare uno scambio messaggi no bloccante sebbene questo impieghi primitive bloccanti.*

Nelle istruzioni della comunicazione selettiva un comando con guardia ha la forma

$$< guard > \rightarrow < statement >$$

dove la guardia ¡guard¿ é costituita da un'espressione booleana B, eventualmente seguita da un'istruzione di scambio messaggi. L'esecuzione di un'istruzione con guardia dipende dalla *regola di valutazione della guardia*, la quale é definita nel modo seguente.

La guardia ha **successo** *se l'espressione booleana é* vera *e l'esecuzione della corrispondente istruzione di scambio messaggi non provocherá ritardo; la guardia* **fallisce** *se l'espressione booleana é* falsa*; temporaneamente la guardia* **non fallisce ma non ha successo** *se l'espressione booleana é vera sebbene l'istruzione di scambio messaggi non possa essere eseguita senza causare ritardo.*

La comunicazione selettiva prevede due tipi di istruzioni composte : l'alternativa e l'iterativa. Nel primo caso, l'istruzione alternativa

if $G_1 \rightarrow S_1$
[] $G_2 \rightarrow S_2$
....
[] $G_n \rightarrow S_n$
endif

viene eseguita valutando dapprima l'espressioni booleane che compaiono all'interno di ciascuna guardia. Successivamente si tenta di eseguire l'istruzione di scambio messaggi, se presente, in corrispondenza di quelle guardie le cui espressioni booleane sono state valutate a vero. Piú esattamente un'istruzione alternativa si esegue nel modo indicato di seguito.

Se almeno una guardia ha successo, una di queste, indicata con G_i, viene selezionata non deterministicamente; l'eventuale istruzione di scambio messaggi in G_i viene eseguita e, quindi, anche la corrispondente istruzione S_i che segue. Se tutte le guardie non hanno successo ne falliscono, l'esecuzione viene ritardata fino a quando almeno una ha successo. Se, infine, tutte le guardie falliscono, il comando abortisce.

Nel secondo caso, l'esecuzione del comando iterativo

do $G_1 \rightarrow S_1$
[] $G_2 \rightarrow S_2$
....
[] $G_n \rightarrow S_n$
enddo

ha quasi lo stesso effetto del precedente con l'unica avvertenza che l'esecuzione viene ripetuta fino a quando tutte le guardie falliscono, dopo di che il comando termina. Piú esattamente si puó affermare quanto segue.

Se almeno una guardia ha successo, una di queste, indicata con G_i, viene selezionata non deterministicamente; l'eventuale istruzione di scambio messaggi in G_i viene eseguita e, quindi, anche la corrispondente istruzione S_i che segue. Se tutte le guardie non hanno successo ne falliscono, l'esecuzione viene ritardata fino a quando almeno una di esse ha successo. La selezione e l'esecuzione di un comando con guardia viene ripetuto fino a quando tutte le guardie falliscono, dando luogo alla terminazione del comando.

Se l'istruzione **send** non puó comparire all'interno di una guardia, la comunicazione selettiva é implementabile in modo semplice e diretto. Un processo in attesa determina quali espressioni booleane nelle guardie sono vere e quindi attende l'arrivo di messaggi che gli permettono l'esecuzione dell'istruzione **receive** all'interno della guardia prescelta.

Se la guardia non contiene nessuna istruzione **receive** allora il processo non viene ritardato. Se entrambe le istruzioni **send** e **receive** possono apparire nelle guardie, l'implementazione é molto piú costosa perché ciascun processo deve negoziare con gli altri processi se puó comunicare i quali, a loro volta, possono trovarsi nel mezzo di una negoziazione. Nel caso, ad esempio, di tre processi che possono eseguire istruzioni di comunicazione selettiva in cui ogni coppia puó comunicare, é necessario stabilire quale coppia puó comunicare mentre il restante processo é costretto ad attendere. Il problema diventa, pertanto, stabilire dei protocolli che risolvono il problema in modo efficiente e senza condurre a situazioni di stallo.

Sfortunatamente, se non si permette che l'istruzione **send** possa apparire nelle guardie, programmare con **send** e **receive** bloccanti puó diventare complesso. Nell'esempio che segue, la sezione di codice del processo *Pipe* relativa all'invio di dati al consumatore, dovrebbe avere nella guardia un'istruzione *send*. Invece, vi compare un'istruzione *receive* che ha lo scopo di chiedere al processo *Pipe* che questo spedisca un messaggio al *Consumatore*, contente i dati da lui richiesti.

La differenza di protocollo utilizzato dal processo *Pipe* a seconda che interagisca col produttore o col consumatore é fuorviante in quanto la relazione *produttore-consumatore* é inerentemente simmetrica e il programma dovrebbe rispecchiare questo fatto.

```
procedure Pipe( PId1, PId2 : ClientId );
varBu : Buffer;
      Msg : Message;
      Dim : Size;
begin
   InitB( Bu );
      do Dim < BufferSize; receive( PId1, Msg ) →
         begin
            PutB( Bu, Msg );
            Dim := Dim + 1
         end
```

```
            [] Dim > 0; receive( PId2, Go ) →
               begin
                  GetB( Bu, Msg );
                  send( PId2, Msg ):
                  Dim := Dim - 1
               end
            endo
      end;
```

La relazione *client/server*, invece, é inerentemente asimmetrica in quanto il processo server si viene a trovare al centro delle diverse richieste dei clienti, rispetto alle quali risponde con azioni adeguate.

3.5 Chiamata di Procedura Remota

Le primitive di scambio messaggi della sezione precedenti sono sufficienti a programmare ogni tipo di interazione fra processi. L'interazione client/server, comunque, richiede che entrambi i processi eseguano due istruzioni di scambio messaggi: *il cliente una* send *seguita da una* receive, e *il server una* receive *seguita da una* send. La larga diffusione di questo tipo di interazione ha portato all'introduzione di costrutti specifici noti, generalmente, col nome di **chiamata di procedura remota**. La motivazione di questa terminologia é presto detta: *il cliente chiama una procedura che viene eseguita dal server su una macchina potenzialmente remota*

In questo caso i clienti interagiscono con il server per mezzo di un'istruzione **call**, la quale si presenta in una forma simile a quella utilizzata per le chiamate di procedura nei linguaggi sequenziali. Si ha, ad esempio,

$$\mathbf{call} Service(Values, Results)$$

Il servizio *Service* rappresenta piú propriamente il nome di un canale il quale, nel caso della denominazione diretta designa il processo server, mentre se é una porta o una mailbox *Service* denota il tipo di servizio richiesto.

3.6 ESERCIZI

Gli esercizi che seguono si basano tutti sulla *comunicazione selettiva di Hoare* dove le richieste di servizio dei clienti sono realizzate tramite chiamate di procedura remota.

3.6.1 Canali di Comunicazione fra Processi

Si consideri un ambiente di comunicazione fra processi costituito da una struttura che sostiene un insieme di canali generalizzati di comunicazione. Ogni *canale* é univocamente determinato da un *punto di ascolto* cui i processi *Clienti* possono connettersi per inviare messaggi ad altri processi che possono ascoltarli, detti *Server*. Si supponga, inoltre, che al piú un solo processo per volta possa ascoltare un dato valore trasmesso[19].

Si realizzi tale scenario mediante uno schema di comunicazione fra processi che utilizza primitive di *scambio messaggi su porte*, definendo un canale come *riferimento* al corrispondente punto di ascolto. Si strutturi quest'ultimo, e lo si identifichi col termine socket, in modo tale da mantenere traccia sia dei messaggi ancora presenti sul canale[20], sia dei processi in ascolto che attendono l'arrivo di un messaggio dal canale. Si supponga che i processi possano accedere ad un canale di comunicazione mediante le seguenti procedure:

- Connect(**var** Chn : *Channel*; Msg : *Message*), che permette di attivare la connessione sul canale inviando un messaggio;
- Liste(**var** : *Channel* ; **var** Msg : Message), che realizza la ricezione del messaggio presente nel canale.

A tal fine si proceda secondo lo schema indicato di seguito.

- Si definisca una porta per ciascun tipo di operazione sul canale (*LPort* , *CPort*).
- Si definiscano i canali come riferimenti a punti di ascolto, strutturando quest'ultimi secondo le direttive precedentemente assegnate.
- Si implementi il processo Server di gestione dei canali mediante la procedura *SocketDaemon* il cui codice utilizza istruzioni di scambio messaggi con guardia le quali possono comparire solo all'interno di istruzioni composte **do...enddo** e **if...endif**[21].

Per quanto riguarda le operazioni *Listen* e *Connect* sui canali é sufficiente indicare in che modo le primitive **send** e **receive** utilizzano i messaggi all'interno di esse.

```
type Channel = ↑Socket;

Socket = record
              Bu : Buffer;
              Qu : Queue
         end;

procedure GetPid( var PId : Process ); external;
```

[19] nel senso che, se piú processi possono ricevere un certo valore, il primo che effettivamente lo riceve, lo distrugge
[20] in attesa di essere ricevuti
[21] *comunicazione selettiva di Hoare*

```
procedure Insert( PId : Process; var Qu : Queue ); external;
procedure Remove( var PId : Process; var Qu : Queue ); external;
function Awaiting( Qu : Queue ) : boolean; external;
function Awaiting IsEmpty( Bu : Buffer ) : boolean; external;
procedure Deposit( var Bu : Buffer; Msg : Message ); external;
procedure Fetch( var Bu : Buffer; var Msg : Message ); external;
procedure AwakeSockets; external;

procedure Connect( var Chn : Channel; Msg : Message );
begin
   send( CPort, Chn, Msg )
end;

procedure Listen( var Chn : Channel; var Msg : Message );
var Curr : Process;
begin
   GetPId( Curr );
   send( LPort, Curr, Chn );
   receive( SocketDaemon, Msg );
end
procedure SocketDaemon;
var Chn : Channel;
      Msg : Message;
      PId : Process;
begin
   AwakeSockets;
   do true; receive( CPort, Chn, Msg ) →
      with Chn↑ do
         if Awaiting( Qu ) →
            begin
               Remove( Qu, PId );
               send( PId, MSg )
            end
         [] not Awaiting( Qu ) → Deposit( Bu, Msg )
         endif
   [] true; receive( LPort, PId, Chn ) →
      with Chn↑ do
         if IsEmpty( Bu ) → Insert( Qu, PId )
         [] not IsEmpty( Bu ) →
            begin
               Fetch( Bu, Msg );
               send( PId, Msg )
            end
         endif     enddoend;
```

3.6.2 Gestione di una Stampante

Si consideri il problema di realizzare la gestione di una stampante il cui driver utilizza una ipotetica primitiva *WriteB*(Bu) che scrive il contenuto di un buffer come linea di stampa. Utilizzando lo schema di comunicazione fra processi che impiega primitive di *scambio messaggi su porte*,

- si definiscano le porte *Lpr* e *Done*, la prima utilizzata per ottenere l'accesso esclusivo alla stampante e la seconda per rilasciare il controllo esclusivo della stampante;
- si realizzino le procedure *Request* e *Release*, in termini di **send/receive**, la prima avente l'effetto di servire solo le successive richieste del cliente *ClientId*, ritardando tutte le richieste degli altri clienti, la seconda che permette a *ClientId* di rilasciare la stampante;
- si implementi il processo Server di gestione della Stampante mediante la procedura *Printer* il cui codice utilizza istruzioni di scambio messaggi con guardia le quali possono comparire solo all'interno di istruzioni composte **do...enddo** e **if...endif** (*comunicazione selettiva di Hoare*).

```
type Queue = record
            Front : Process;
            Rear : Process
        end;

    LprPort = port
            Pid : Process;
            Bu : Buffer
        end;

var Req : LprPort;
    Lpr : LprPort;

procedure GetPid( var Pid : Process ); external;
procedure WriteB( var Bu : Buffer ); external;

procedure Request( var Bu : Buffer );
var Curr : Process;
        Go : boolean;
begin
    GetPid( Curr );
    send( Lpr, Curr, Bu );
    receive( Printer, B )
end;

procedure Printer;
var Qu : Queue;
        Jobs : integer;
        Again : boolean;
        Curr : Process;
        Pid : Process;
        Free : boolean;
```

```
        Bu : Buffer;
    begin
        Create( Qu );
        Free := true;
        Jobs := 0;
        do true; receive( Req, Pid ) →
            begin
                Jobs := Jobs + 1;
                if Free then
                    begin
                        Free := false;
                        Curr := Pid;
                        send( Pid, true )
                    end else
                    Insert( Qu, Pid );
                end;
        [] Jobs > 0; receive( Lpr, Pid, Bu ) →
            if ( Curr = Pid ) then
                begin
                    Free := false;
                    WriteB( Bu );
                    send( Pid, true )
                end else
                send( Pid, false );
        [] not Free; receive( Done, Again ) →
            begin
                Jobs := Jobs - 1;
                if not Empty( Qu ) then
                    begin
                        Remove( Qu, Pid );
                        send( Pid, true )
                    end else
                    Free := true
            end
        endo
    end;
```

3.6.3 Gestione di un Pool di Driver mediante Scambio Messaggi

Si consideri il problema di gestire tramite un *TapeServer* l'accesso in scrittura ad un certo numero (M = 5) di unitá nastro. Il *server* riceve dai processi *clienti* le richieste di scrittura[22] mediante la chiamata di procedura seguente

$$\textbf{procedure} Dump(\textbf{var} Bu : Buffer; Len : \textbf{integer}; \textbf{var} M : \textbf{integer})$$

accodando la richiesta al *Driver* dell'unitá nastro che é in grado di soddisfare la richiesta nel minor tempo possibile, il cui identificatore *M* viene restituito alla procedura chiamante Dump.

[22] *numero dei blocchi* e *indirizzo del buffer* in cui tali blocchi risiedono

Si implementi la procedura *TapeServer* che realizza la gestione dei driver delle unitá nastro utilizzando le primitive di scambio messaggi **send** e **receive** su porte con i costrutti di *comunicazione selettiva* di Hoare **do...enddo** e **if..endif**. Si scriva l'algoritmo ipotizzando che i processi clienti spediscano le loro richieste alla porta *Tape*[23] supponendo che i tali processi attendano la notifica dell'identificatore dell'unitá nastro su cui é stato eseguito il dump.

Si commentino adeguatamente le strutture dati introdotte per gestire appropriatamente le risorse nastro. In particolare, é necessario mantenere per ogni unitá l'informazione relativa al numero di blocchi pendenti in scrittura, ricevendo da ciascuna di esse il numero di blocchi scritto nell'ultima operazione.

```
type DriverName = 1 .. 5;

        TapePort = port
                       Bu : ↑Buffer;
                       Len : integer;
                       Pid : Process
                   end;

var Tape : TapePort;
procedure GetPid( var Pid : Process ); external;
function Select( var M : integer ) : DriverName; external;
procedure MakeDump( var Bu : Buffer; Len : integer ); external;

procedure Dump( var Bu : Buffer; L : integer; var M : DriverName );
var Curr : Process;
begin
    GetPid( Curr );
    send( Tape, Bu, L, Curr );
    receive( TapeServer, M )
end;

procedure Driver( Q : DriverName );
begin
    repeat
        receive( TapeServer, Bu, Len );
        MakeDump( Bu, Len );
        send( TapeServer, Len )
        until false
end;

procedure TapeServer;
var K : DriverName;
        M : integer;
        Len : integer;
        D : array[ DriverName ] of integer;
begin
```

[23] la quale, pertanto, si comporta da unitá nastro virtuale

```
        for K := 1 to 5 do D[ K ] := 0;
        M := 0;
        do true; receive( Tape, Bu, Len, Pid ) →
            begin
                K := Select( M );
                D[ K ] := D[ K ] + Len;
                send( K, Bu, Len );
                send( Pid, K )
            end;
        [] D[ 1 ] > 0; receive( 1, Len ) → D[ 1 ] := D[ 1 ] - Len;
        [] D[ 2 ] > 0; receive( 2, Len ) → D[ 2 ] := D[ 2 ] - Len;
        [] D[ 3 ] > 0; receive( 3, Len ) → D[ 3 ] := D[ 3 ] - Len;
        [] D[ 4 ] > 0; receive( 4, Len ) → D[ 4 ] := D[ 4 ] - Len;
        [] D[ 5 ] > 0; receive( 5, Len ) → D[ 5 ] := D[ 5 ] - Len;
        endo
    end;
```

3.6.4 Autorizzazione d'accesso mediante Pagine Gialle

Si consideri la gestione dell'accesso ad un sistema distribuito in rete locale per utenti autorizzati. Si supponga che ciascun processo possa richiedere l'uso di un processore mediante la funzione *Login* la quale accetta come ingresso due stringhe che rappresentano, rispettivamente, il *nome* e la *chiave* dell'utente che vuole utilizzare il processore e che restituisce, in uscita, un codice numerico che identifica univocamente l'utente[24]. Tale funzione assume il valore true se l'utente é autorizzato ad accedere al processore, false altrimenti. Inoltre, alcuni processi[25] possono permettere l'autorizzazione a nuovi utenti di accedere alla rete locale di processori, invocando la procedura *YPBind* i cui paramentri sono definiti analogamente al caso precedente.

Tenendo presente lo schema precedente[26], si supponga che siano disponibili le seguenti operazioni

```
procedure GetPid( var Proc : Process );
procedure GetUid( Name : String; var User : UserId );
procedure SetUid( Name, Passwd : String; var Uid : UserId );
function Authorized( Name : String; Uid : UserId ) : boolean;
```

la prima per estrarre il Pid del processo corrente, la seconda che associa al nome simbolico dell'utente il suo codice[27], la terza che introduce un nuovo utente nel data-base degli utenti e la quarta che verifica la correttezza della *chiave di accesso*.

Utilizzando lo schema di comunicazione fra processi che impiega primitive di scambio messaggi su porte,

- si definiscano le porte *Acct* e *Auth*, la prima utilizzata per ottenere l'accesso al sistema e la seconda per richiedere nuove autorizzazioni;
- si realizzino le procedure *Login* e *YPBind*, precedentemente citate, in termini di **send/receive**;

[24] noto come *UId*, o *UserIdentifier*
[25] noti come *processi gestori*
[26] cui verrá fatto riferimento col termine *Pagine Gialle*
[27] il cui valore é -1 se tale associazione non é soddisfacibile confrontando il data-base corrente degli utenti

- si implementi il processo Server di gestione delle Pagine Gialle mediante la procedura *YPServer* il cui codice utilizza istruzioni di scambio messaggi con guardia le quali possono comparire solo all'interno di istruzioni composte **do...enddo** e **if...endif** (*comunicazione selettiva di Hoare*).

```
type Process = 0 .. MaxProcess;
     UserId = 0 .. MaxUser;

     String = record
          Len : 0 .. MaxLen;
          Data : array[ 1 .. MaxLen ] of char
       end;

     Acct : port
          Name : String;
          Passwd : String;
          PId : Process
       end;

     Auth : port
          Name : String;
          Passwd : String;
          PId : Process
       end;

procedure GetPid( var Proc : Process ); external;
procedure GetUid( Name : String; var User : UserId ); external;
procedure SetUid( Name, Passwd : String; var Uid : UserId ); external;
function Authorized( Name : String; Uid : UserId ) : boolean; external;
procedure YPInit; external;
procedure CShell; external;

function Login( Name, Passwd : String; var UId : UserId ) : boolean;
var Curr : Process;
    LogF : boolean;
begin
   GetPid( Curr );
   send( Acct, Name, Passwd, Curr );
   receive( YPServer, UId, LogF );
   Login := LogF
end;

procedure YPBind( Name, Passwd : String; var UId : UserId );
var Curr : Process;
begin
   GetPid( Curr );
   send( Auth, Name, Passwd, Curr );
   receive( YPServer, UId );
```

```
end;

procedure YPServer;
var Passwd : String;
      Name : String;
      PId : Process;
      UId : UserId;
begin
   YPInit;
   do true; receive( Acct, Name, Passwd, PId ) →
      begin
         GetUid( Name, UId );
         if not Authorized( UId, Passwd ) then UId := -1;
         send( PId, UId )
      end;
   [] true; receive( Auth, Name, Passwd, PId ) →
      begin
         SetUid( Name, Passwd, UId );
         send( PId, UId )
      end;
   enddo;
end;

procedure YPPassword( var UId : UserId );
var Passwd : String;
      Name : String;
begin
   write('UserName : '); readln( Name );
   write('Password : '); readln( Passwd );
   YPBind( Name, Passwd, Uid )
end;

procedure RemoteShell;
var Passwd : String;
      Name : String;
begin
   repeat
      write('Login : '); readln( Name );
      write('Password : '); readln( Passwd );
      if Login( Name, Passwd, Uid ) then CShell;
   until UId ¿= 0
end;
```

3.6.5 Gestione della Memoria Virtuale

Si consideri il problema[28] di gestire la lista dei *frames*, resi disponibili dallo swap device, per la realizzazione della Memoria Virtuale costituita da un certo numero N di frames. Si supponga che ciascun processo possa chiedere un numero A di frames mediante la procedura

$$Allocate(A : \textbf{integer})$$

e ne possa rilasciare un numero B utilizzando la procedura

$$Release(B : \textbf{integer})$$

le quali danno luogo ad un protocollo d'utilizzo della memoria virtuale con allocazione parziale.

La gestione della Memoria Virtuale é ottenuta utilizzando la variabile F per identificare il numero corrente di frames ancora disponibili; questa viene incrementata e decrementata in accordo con le richieste, rispettivamente, di rilascio e allocazione, fatte dai processi. Se una richiesta di allocazione non puó essere soddisfatta, in quanto il numero di frames disponibili é inferiore alla richiesta fatta dal processo, questo viene sospeso ed inserito in una coda ordinata per numero di frames richiesti[29]. A tale proposito si suppongano disponibili le seguenti operazioni

```
function Empty( var Qu : Queue ) : boolean;
procedure Insert( var Qu : Queue; PId : Process; K : integer );
procedure Remove( var Qu : Queue; var PId : Process; var K : integer );
```

la prima per verificare se la coda é vuota, la seconda per inserire il processo in accordo ai valori crescenti dell'ultimo parametro, la terza per estrarre il primo processo della coda.

Facendo riferimento ad uno schema di comunicazione fra processi che impiega primitive di *scambio messaggi su porte*,

- si definiscano le porte *APort* e *RPort* utilizzate dai processi per inviare le richieste di allocazione e di rilascio di frames;
- si realizzino le procedure *Allocate* e *Release*, precedentemente citate, in termini di **send/receive**;
- si implementi il processo Server di gestione della Memoria Virtuale mediante la procedura *PageDaemon* il cui codice utilizza istruzioni di scambio messaggi con guardia le quali possono comparire solo all'interno di istruzioni composte **do...enddo** e **if...endif** (*comunicazione selettiva di Hoare*).

```
type Process = 0 .. MaxProcess;

          Queue = record
                    Front : ↑Descriptor;
                    Rear : ↑Descriptor
               end;

          APort : port
```

[28] 31 ottobre 89, 16 ottobre 90

[29] cosicché il primo processo della coda é quello che ha fatto la minore richiesta di frames

```
            PId : Process;
            A : integer
        end;

        RPort : port
            B : integer
        end;

procedure Create( var Qu : Queue ); external;
procedure Insert( var Qu : Queue; Pid : Process; L : integer ); external;
procedure Remove( var Qu : Queue; var Pid : Process; var L : integer ); external;
function Empty( var Qu : Queue ) : boolean; external;
procedure GetPid( var Pid : Pocess ); external;

procedure Allocate( A : integer );
var Done : boolean;
        Proc : Process;
begin
    GetPid( Proc );
    send( APort, Proc, A );
    receive( PageDaemon, Done )
end;

procedure Release( B : integer );
begin
    send( RPort, B )
end;

procedure PageDaemon;
var PId : Process;
        Que : Queue;
        F : integer;
        K : integer;
begin
    F := N;
    Create( Que );
    do F > 0; receive( APort, PId, K ) →
        if F >= K →
            begin
                F := F - K;
                send( PId, true )
            end
        [] F < K → Insert( Que, PId, K )
        endif
    [] F < N; receive( RPort, K ) →
            begin
                F := F + K;
```

```
                if not Empty( Que ) then
                   begin
                      Remove( Que, PId, K );
                      if F >= K →
                         begin
                            F := F - K;
                            send( PId, true )
                         end
                      [] F < K → Insert( Que, PId, K )
                      endif
                   end
             end
       enddo
    end;
```

3.6.6 Gestione di un File-System distribuito

Si consideri il problema di gestire tramite un *NFSDaemon* l'accesso in scrittura ad un certo numero (T = 4) di dischi, ciascuno residente su un elaboratore diverso ed identificato con la variabile intera U. Il gestore *NFSDaemon* riceve dai processi clienti le richieste di accesso[30] mediante la chiamata di procedura

$$\textbf{procedure} Write(\textbf{var} Bu : Buffer; D : Disk; \textbf{var} B : Block)$$

passando le richieste al *Server* del disco corrispondente, in esecuzione sull'appropriata CPU. Tale richiesta viene gestita in modo che venga restituito alla procedura chiamante Write l'indirizzo del primo blocco su disco contenente i dati del buffer copiati.

Si implementi la procedura NFSDaemon che realizza la gestione del file-system distribuito utilizzando le primitive di scambio messaggi **send** e **receive** su porte con i costrutti di *comunicazione selettiva* di Hoare **do...enddo** e **if..endif**. Si scriva l'algoritmo ipotizzando che i processi clienti spediscano le loro richieste alla porta *NFS* [31] e supponendo che tali processi attendano la notifica del blocco su disco a partire dal quale é stato trasferito il contenuto del buffer.

Si commentino adeguatamente le strutture dati introdotte per gestire appropriatamente il sistema NFS. In particolare, é necessario conoscere l'identificatore della CPU su cui esegue il Server del disco corrispondente.

```
    type DiskServer = 1 .. 4;
          Disk = ( DK0, DK1, DK2, DK3 );
          Block = 0 .. MaxBlock;

          NFSPort = port
                Bu : ↑Buffer;
                Dk : Disk;
                Pid : Process
             end;

    var NFS : NFSPort;
```

[30] *indirizzo del buffer* in cui tali blocchi risiedono assieme all'identificatore del disco su cui trasferirli

[31] la quale, pertanto, si comporta come *file system distribuito*

```
procedure GetPid( var Pid : Process ); external;
procedure WriteB( var Bu : Buffer; var B : Block ); external;

function Find( Dk : Disk ) : DiskServer;
begin
   case Dk of
      DK0 : Find := 1;
      DK1 : Find := 2;
      DK2 : Find := 3;
      DK3 : Find := 4
   end
end;

procedure Write( var Bu : Buffer; D : Disk; var B : Block );
var Curr : Process;
begin
   GetPid( Curr );
   send( NFS, Bu, D, Curr );
   receive( NFSDaemon, B )
end;

procedure Driver( U : DiskServer );
begin
   repeat
      receive( NFSDaemon, Bu );
      WriteB( Bu, B );
      send( NFSDaemon, B, Pid )
   until false
end;

procedure NFSDaemon;
var U : DiskServer;
      B : Block;
      W : array[ DiskServer ] of integer;
begin
   for U := 1 to 4 do W[ U ] := 0;
   do true; receive( NFS, Bu, Dk, Pid ) →
      begin
         U := Find( Dk );
         W[ U ] := W[ U ] + 1;
         send( U, Bu, Pid )
      end;
   [] W[ 1 ] > 0; receive( 1, B, Pid ) →
      begin
         W[ 1 ] := W[ 1 ] - 1;
         send( Pid, B )
```

```
        end;
    [] W[ 2 ] > 0; receive( 2, B, Pid ) →
        begin
            W[ 2 ] := W[ 2 ] - 1;
            send( Pid, B )
        end;
    [] W[ 3 ] > 0; receive( 3, B, Pid ) →
        begin
            W[ 3 ] := W[ 3 ] - 1;
            send( Pid, B )
        end;
    [] W[ 4 ] > 0; receive( 4, B, Pid ) →
        begin
            W[ 4 ] := W[ 4 ] - 1;
            send( Pid, B )
        end;
    endo
end;
```

3.6.7 Filosofi al Ristorante

Cinque amici filosofi si recano a pranzo in un ristorante per meditare su importanti problemi metodologici e, nel contempo, mangiare un buon piatto di spaghetti. Si siedono attorno ad un tavolo e, per poter pensare con maggior profitto, il ristorante mette a disposizione per ciascuno di essi un cameriere. Ogni filosofo alterna la *meditazione* con il *pranzo* per il quale utilizza il proprio cameriere, seguendo lo schema riportato oltre:

```
procedure Philosophe( P : Thinker[32] );
begin
    repeat
        Think;
        Beckon;
        Eat ;
        GiveUp( P )
    forever
end
```

Per poter mangiare gli spaghetti il filosofo deve disporre di due forchette mentre le forchette utilizzabili sono soltanto cinque. Come conseguenza di ció, ciascun filosofo puó mangiare solamente se nessuno dei suoi due vicini di tavola sta mangiando. Quando un filosofo desidera mangiare fa un cenno[33] al proprio cameriere che attende che siano disponibili entrambe le forchette ai lati del piatto e, quindi, le passa al filosofo che inizia a mangiare. Successivamente, rinunciando[34] a continuare, riprende a pensare dopo aver riposto le forchette ai lati del proprio piatto.

Utilizzando lo schema di comunicazione fra processi che impiega primitive di *scambio messaggi su porte*,

---

[33] cioé, esegue la procedura *Beckon*
[34] per cui esegue la procedura *GiveUp*

- si introduca la porta *Done* utilizzata da ciascun filosofo per rilasciare le forchette ai lati del proprio piatto;
- si definisca ciascun cameriere mediante un canale a comunicazione diretta;
- si realizzino le procedure *Beckon* e *GiveUp* in termini di operazioni di scambio messaggi;
- si costruiscano le procedure *Serve* e *PutDown* come azioni di base eseguite la prima da ciascun cameriere per assegnare le forchette al relativo filosofo, la seconda eseguita da ogni filosofo quando termina di mangiare;
- si implementi il processo di gestione dei camerieri mediante la procedura *Waiter* il cui codice utilizza istruzioni di scambio messaggi con guardia le quali possono comparire solo all'interno di istruzioni composte **do...enddo** e **if...endif** (*comunicazione selettiva di Hoare*).

Utilizzando l'estensione del PASCAL proposta si implementi un processo per ciascun filosofo ıp (0 ¡= p ¡= 4) in modo tale da prevenire situazioni di deadlock.

```
const N = 5;

type Thinker = 0 .. N - 1;

procedure Beckon( P : Thinker);
var CanEat : boolean;
begin
    send( Waiter, true );
    receive( Waiter, CanEat )
end;

procedure GiveUp( Pid : Thinker);
begin
    send( Done, Pid )
end;

procedure Waiter;
type AnyFork = 0 .. 2;
        Table = array[ Thinker ] of AnyFork;

var K : Thinker;
        M : integer;
        Req : boolean;
        Forks : Table; { Forchette non utilizzate per ciascun piatto }

    procedure Serve( Pid : Thinker );
    var St : AnyFork;
            Nd : AnyFork;
    begin
        St := Pid;
        Nd := ( Nd + 1 ) mod N;
```

```
        Forks[ St ] := Forks[ St ] - 1;
        Forks[ Nd ] := Forks[ Nd ] - 1
    end;

    procedure PutDown( Pid : Thinker );
    var St : AnyFork;
            Nd : AnyFork;
    begin
        St := Pid;
        Nd := ( Nd + 1 ) mod N;
        Forks[ St ] := Forks[ St ] + 1;
        Forks[ Nd ] := Forks[ Nd ] + 1
    end;

begin
    for K := 0 to N - 1 do Forks[ K ] := 2;
    M := 0;
    do ( Forks[ 0 ] = 2 ); receive( 0, Req ) →
        begin
            Serve( 0 );
            M := M + 1;
            send( Pid, true )
        end;
    [] ( Forks[ 1 ] = 2 ); receive( 1, Req ) →
        begin
            Serve( 1 );
            M := M + 1;
            send( Pid, true )
        end;
    [] ( Forks[ 2 ] = 2 ); receive( 2, Req ) →
        begin
            Serve( 2 );
            M := M + 1;
            send( Pid, true )
        end;
    [] ( Forks[ 3 ] = 2 ); receive( 3, Req ) →
        begin
            Serve( 3 );
            M := M + 1;
            send( Pid, true )
        end;
    [] ( Forks[ 4 ] = 2 ); receive( 4, Req ) →
        begin
            Serve( 4 );
            M := M + 1;
            send( Pid, true )
        end;
```

```
    [] ( M > 0 ); receive( Done, Pid ) → PutDown( Pid )
    [] enddo
end;

procedure Philosophe( P : Thinker );
begin
    repeat
        Think;
        Beckon( P );
        Eat;
        GiveUp( P )
    until false
end;
```

3.6.8 Il Problema dei tre fumatori

Si consideri un sistema costituito da tre processi fumatori e un processo agente. Ciascun fumatore ripete indefinitivamente le due azioni seguenti : prepara una sigaretta e poi la fuma. Per preparare una sigaretta sono necessari due ingredienti : tabacco e carta; inoltre, é necessario un fiammifero per accenderla. Ciascuno dei tre processi fumatori dispone soltanto di uno degli ingredienti: il primo ha solo il tabacco, il secondo la carta mentre il terzo dispone solo dei fiammiferi.

Per poter permettere ai processi fumatori di fumare le sigarette é presente un processo agente che é in grado di fornire tutti gli ingredienti necessari. L'unica limitazione é data dal fatto che ne fornisce soltanto due per volta in modo da permettere al processo che dispone dell'elemento mancante di preparare e fumare la propria sigaretta. Appena il fumatore ha prelevato gli ingredienti di cui abbisogna, notifica il fatto al processo agente in modo da permettergli di mettere a disposizione dei fumatori un'altra coppia di ingredienti.

- Si implementi ciascun processo fumatore come un cliente che notifica al processo agente di quale ingrediente puó disporre e, quindi, attendendo i mancanti, si sospende prima di iniziare a fumare la sigaretta.
- Si implementi il processo agente come un processo server che accetta sulle opportune porte le richieste dei processi fumatori attivando di volta in volta il processo in grado di fumare la sigaretta. Si supponga che sia disponibile una funzione in grado di generare in modo random gli elementi prodotti dall'agente.

Si risolva il problema utilizzando il metodo della comunicazione selettiva di Hoare, cioé mediante l'impiego di istruzioni di scambio messaggi controllate da guardie.

```
type Ingredient = ( Tobacco, Paper, Match );

procedure Prepare; external;
procedure Smoke; external;
function GenIngr : Ingredient; external;

procedure Request( Ingr : Ingredient );
var Done : boolean;
```

```
begin
    case Ingr of
        Tobacco : send( TobaccoPort, true );
        Paper : send( PaperPort, true );
        Match : send( MatchPort, true )
    end;
    receive( Agent, Done )
end;

procedure Continue( Ingr : Ingredient );
var Done : boolean;
begin
    send( Smoker( Ingr ), TRUE );
    receive( Smoker( Ingr ), Done )
end;

procedure Smoker( Ingr : Ingredient );
begin
    repeat
        Request( Ingr );
        Prepare;
        send( Agent, true );
        Smoke
    until false
end;

procedure Agent;
var Ingr : Ingredient;
        Any : boolean;
begin
    Ingr := GenIngr;
    do Ingr = Tobacco; receive( TobaccoPort, Any ) →
        begin
            Continue( Tobacco );
            Ingr := GenIngr
        end;
    [] Ingr = Paper; receive( PaperPort, Any ) →
        begin
            Continue( Paper );
            Ingr := GenIngr
        end;
    [] Ingr = Match; receive( MatchPort, Any ) →
        begin
            Continue( Match );
            Ingr := GenIngr
        end;
    endo
```

```
end;
```

3.6.9 Implementazione di Pipe mediante Buffer

Si supponga di voler intermediare la comunicazione diretta fra una coppia di processi *Produttore/Consumatore* con un buffer di memorizzazione temporanea dei dati generati dal produttore prima che vengano utilizzati dal consumatore per ottimizzare l'impiego delle risorse. In questo modo l'usuale operazione "—"di Unix viene sostituita con un'operazione *Deposit* per il produttore e un'operazione *Fetch* per il consumatore seguendo lo schema riportato di seguito:

```
procedure Producer ;             procedure Consumer;
var Msg : Message;               var Msg : Message;
begin                            begin
   repeat                           repeat
      Produce( Msg );                  Fetch( Msg );
      Deposit( Msg );                  Consume( Msg );
   forever                          forever
end                              end
```

dove sono state evidenziate le operazioni di scambio messaggi rispetto alle operazioni proprie effettuate dai singoli processi.

Utilizzando lo schema di comunicazione fra processi che impiega primitive di *scambio messaggi su porte*,

- si introducano le porte *PPort* e *CPort* utilizzate, rispettivamente, dai processi produttore e consumatore;
- si realizzino le procedure *Deposit* e *Fetch* in termini di operazioni di scambio messaggi;
- si implementino le procedure *PutB* e *GetB* aventi un solo parametro ciascuna, che rappresenta il messaggio inserito o estratto, per la gestione esplicita del buffer;
- si implementi il processo di gestione del buffer di comunicazione mediante la procedura ıBuffer il cui codice utilizza istruzioni di scambio messaggi con guardia le quali possono comparire solo all'interno di istruzioni composte **do...enddo** e **if...endif** (*comunicazione selettiva di Hoare*).

Si tenga presente che il produttore non puó introdurre messaggi se il buffer é pieno cosí come il consumatore non ne puó estrarre se il buffer é vuoto. L'eventuale attesa del verificarsi delle condizioni precedenti da parte dei processi interessati non deve essere gestita mediante code.

```
type Process = 0 .. MaxProcess;

var CPort : port
               PId : Process
            end;
    PPort : port
               Msg : Message
            end;

procedure GetPid( var Proc : Pocess ); external;
```

```
procedure Deposit( Msg : Message )
begin
    send( PPort, Msg )
end;

procedure Fetch( var Msg : Message )
var PId : Process
begin
    send( CPort, Pid )
    receive( Buffer,Msg )
end;

procedure Buffer;
type Index = 0 .. BufferSize;
        BufferData = array[ Index ] of Message;

var PId : Process
        Head : Index;
        Tail : Index;
        Data : BufferData;
        L : Index;

    procedure Put( Msg : Message );
    begin
        Data[ Tail ] := Msg;
        Tail := ( Tail + 1 ) mod BufferSize
    end;

    procedure Get( Msg : Message );
    begin
        Msg := Data[ Head ];
        Head := ( Head + 1 ) mod BufferSize
    end;

begin
    L := BufferSize;
    do L > 0; receive( CPort, Pid )
        begin
            Get( Msg );
            L := L - 1;
            send( Pid, Msg )
        end
    do L < BufferSize; receive( RPort, Msg ) →
        begin
            Put( Msg );
            L := L + 1;
```

```
            end
         endo
      end;
```

3.6.10 Gestione della Frammentazione Esterna in un file-system

Si consideri il problema di mantenere la lista deiblocchi liberi all'interno di una partizione di disco per la gestione del file-system su di essa. Detto N il numero massimo di blocchi disponibili, si supponga che ogni processo che deve modificare un file possa acquisire e rilasciare blocchi mediante le procedure

Request (A : **integer**)
Release (A : **integer**)

essendo A un parametro preventivamente determinato, dando luogo all'ovvio protocollo d'utilizzo dello spazio su disco.

La gestione dei blocchi liberi é ottenuta utilizzando la variabile F per identificare il numero di blocchi attualmente disponibili; questa viene incrementata e decrementata in accordo con le richieste, rispettivamente, di rilascio e acquisizione, fatte dai processi. Se una richiesta di allocazione non puó essere soddisfatta, in quanto il numero di blocchi disponibili é inferiore alla richiesta fatta dal processo, questo viene sospeso ed inserito in una coda ordinata per numero di blocchi richiesti[35]. A tale proposito si suppongano disponibili le seguenti operazioni

function Empty(**var** Qu : Queue) : **boolean**;
procedure Insert(**var** Qu : Queue; PId : Process; K : **integer**);
procedure Remove(**var** Qu : Queue; **var** PId : Process; **var** K : **integer**);

la prima per verificare se la coda é vuota, la seconda per inserire il processo in accordo ai valori crescenti dell'ultimo parametro, la terza per estrarre il primo processo della coda.

Facendo riferimento ad uno schema di comunicazione fra processi che impiega primitive di scambio messaggi su porte,

- si definiscano le porte *Lock* e *Unlock* utilizzate dai processi per inviare le richieste di allocazione e di rilascio di blocchi;
- si realizzino le procedure *Request* e *Release*, precedentemente citate, in termini di **send/receive**;
- si implementi il processo di gestione dei blocchi liberi mediante la procedura *Allocator* il cui codice utilizza istruzioni di scambio messaggi con guardia le quali possono comparire solo all'interno di istruzioni composte **do...enddo** e **if...endif** (*comunicazione selettiva di Hoare*).

3.6.11 Spooling distribuito in rete locale

Si consideri il problema di gestire tramite un *NetSpooler* l'accesso ad un certo numero (L = 5) di stampanti, ciascuna residente su un elaboratore diverso ed identificato con la variabile intera U. Il gestore *NetSpooler* riceve dai processi clienti le richieste di accesso[36] mediante la chiamata di procedura

$$\textbf{procedure} Write(\textbf{var} Bu : Buffer; L : Printer; \textbf{var} B : Block)$$

[35] cosicché il primo processo della coda é quello che ha fatto la minore richiesta di blocchi

[36] *indirizzo del buffer* in cui risiedono i blocchi del file da stampare assieme all'identificatore della stampante su cui stamparli

passando le richieste allo *spooler* della stampante, in esecuzione sull'appropriata CPU. Tale richiesta viene gestita in modo che venga restituito alla procedura chiamante Write l'indirizzo del primo blocco su disco contenente i dati del buffer copiati.

Si implementi la procedura NetServer che realizza la gestione della stampa in rete locale utilizzando le primitive di scambio messaggi **send** e **receive** su porte con i costrutti di *comunicazione selettiva* di Hoare **do...enddo** e **if..endif**. Si scriva l'algoritmo ipotizzando che i processi clienti spediscano le loro richieste alla porta *Lpd*[37] e supponendo che tali processi attendano la notifica dell'avvenuto trasferimento del file da stampare.

```
type Spooler = 1 .. 5;
        Lpr = ( Lp0, Lp1, Lp2, Lp3, Lp4 );
        Block = 0 .. MaxBlock;
        LpdPort = port
        Bu : Buffer;
                Lp : Lpr;
                Pid : Process
            end;

varLpd   :   LpdPort;

procedure GetPid( var Pid : Process ); external;
procedure WriteB( var Bu : Buffer; var B : Block ); external;

function Find( Lp : Lpr ) : Spooler;
begin
    case Lp of
        Lp0 : Find := 1;
        Lp1 : Find := 2;
        Lp2 : Find := 3;
        Lp3 : Find := 4;
        Lp4 : Find := 5
    end
end;

procedure Write( var Bu : Buffer; L : Printer );
varCurr : Process;
        Done : boolean;
begin
    GetPid( Curr );
    send( Lpd, Bu, L, Curr );
    receive( NetSpooler, Done )
end;

procedure Printer( S : Spooler );
begin
    repeat
```

[37] la quale, pertanto, si comporta come *una stampante di rete*

```
        receive( NetSpooler, Bu, Pid );
        WriteB( Bu, B );
        send( NetSpooler, Pid )
    until false
end;

procedure NetSpooler;
varS : Spooler;
        Bu : Buffer;
        W : array[ Spooler ] of integer;
begin
    for S := 1 to 5 do W[ S ] := 0;
    do true; receive( Lpd, Bu, Lp, Pid ) -¿
        begin
            S := Find( Lp );
            W[ S ] := W[ S ] + 1;
            send( S, Bu, Pid )
        end;
        [] W[ 1 ] > 0; receive( 1, Pid ) -¿
            begin
                W[ 1 ] := W[ 1 ] - 1;
                send( Pid, true )
            end;
        [] W[ 2 ] > 0; receive( 2, Pid ) -¿
            begin
                W[ 2 ] := W[ 2 ] - 1;
                send( Pid, true )
            end;
        [] W[ 3 ] > 0; receive( 3, Pid ) -¿
            begin
                W[ 3 ] := W[ 3 ] - 1;
                send( Pid, true )
            end;
        [] W[ 4 ] > 0; receive( 4, Pid ) -¿
            begin
                W[ 4 ] := W[ 4 ] - 1;
                send( Pid, true )
            end;
        [] W[ 5 ] > 0; receive( 5, Pid ) -¿
            begin
                W[ 5 ] := W[ 5 ] - 1;
                send( Pid, true )
            end;
        endo
end;
```

3.6.12 Gestione della memoria heap

Si supponga di voler gestire uno spazio comune di memoria per tutti i processi che ne fanno richiesta. Tale spazio puó essere assegnato in unitá contigue di dimensione fissata (blocchi). Ciascun processo puó fare richiesta di un numero non superiore ad N di blocchi mediante la procedura *MAlloc* e, se esiste la quantitá richiesta di memoria questa viene assegnata passando al processo l'indirizzo A del primo blocco. I blocchi non ulteriormente utilizzati vengono rilasciati eseguendo l'operazione *DeAlloc.* Se la richiesta non puó essere momentaneamente soddisfatta il processo viene posto in attesa avendo cura di risvegliare, quando nuovi blocchi vengono rilasciati, quei processi che hanno fatto minor richiesta.

Utilizzando lo schema di comunicazione fra processi che impiega primitive di *scambio messaggi su porte*,

- si introducano le porte *Heap* e *Done* utilizzate, rispettivamente, da ciascun processo per effettuare richieste di allocazione e rilascio di memoria;
- si realizzino le procedure *MAlloc* e *DeAlloc* in termini di operazioni di scambio messaggi;
- si implementi il processo di gestione della memoria comune mediante la procedura *HeapHandler* il cui codice utilizza istruzioni di scambio messaggi con guardia le quali possono comparire solo all'interno di istruzioni composte **do...enddo** e **if...endif** (*comunicazione selettiva di Hoare*).

Si realizzi la gestione dello heap mediante una variabile F che tenga conto del numero di blocchi correntemente disponibili. Si supponga, inoltre, che sia disponibile una procedura FreeMem a due parametri tale che, assegnato il numero L di blocchi contigui, restituisca l'indirizzo del primo.

```
type Process = 0 .. MaxProcess;

var Heap : port
              PId : Process;
              L : integer
           end;

    Done : port
              L : integer
           end;

procedure Create( var Qu : Queue ); external;
procedure Insert( var Qu : Queue; Proc : Process; L : integer ); external;
procedure Remove( var Qu : Queue; var Proc : Process; var L : integer ); external;
function Empty( var Qu : Queue ) : boolean; external;
procedure GetPid( var Proc : Pocess ); external;
procedure FreeMem( L : integer ; var A : Address ); external;

procedure MAlloc( L : integer ; var A : Address);
var Proc : Process;
begin
   GetPid( Proc );
   send( Heap, Proc, L );
   receive( Allocator, A )
```

```
end;

procedure DeAlloc( L : integer );
begin
   send( Done, L )
end;

procedure HeapHandler;
varPId : Process;
      F : integer;
      L : integer;
      Qu : Queue;
begin
   F := N;
   Create( Qu );
   do F > 0; receive( Heap, PId, L ) →
      if L >= F then
         begin
            F := F - L;
            FreeMem( L, A )
            send( PId, A )
         end else
         Insert( Qu, PId, L )
      [] F < N; receive( Done, L ) →
         begin
            F := F + L;
            if not Empty( Qu ) then
               begin
                  Remove( Qu, PId, L );
                  if L <= F then
                     begin
                        F := F - L;
                        FreeMem( L, A );
                        send( PId, A )
                     end else
                     Insert( Qu, PId, L )
               end
         end
      enddo
end;
```