

Il Modello a scambio di messaggi

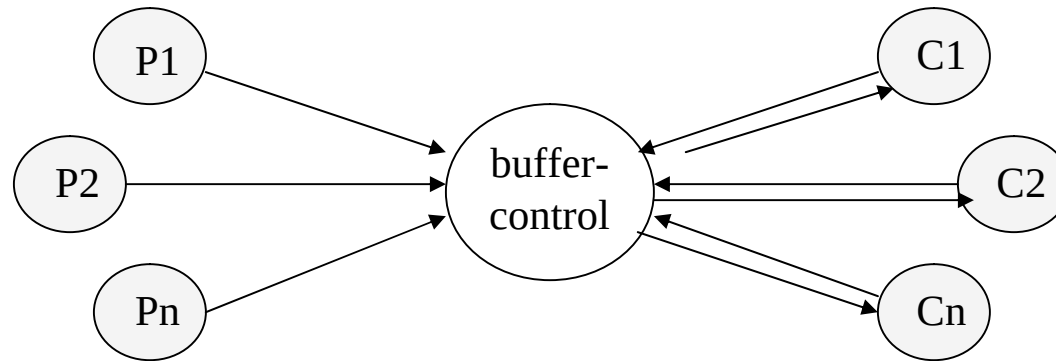
PRIMITIVE PER LO SCAMBIO DI MESSAGGI

- Un messaggio si può considerare costituito da: *origine*, *destinazione* e *contenuto*

```
typedef struct{  
    ... origine;  
    ... destinazione;  
    ... contenuto;  
} messaggio;
```

- Nel caso più semplice si può supporre che:
 - ad ogni processo sia associata una coda per i messaggi in arrivo
 - le primitive di comunicazione usate dai processi sono:
send(m) **receive(m)** (dove **message m;**)
- La primitiva **send(m)** inserisce il messaggio m nella coda del destinatario
- La primitiva **receive(m)** preleva un messaggio dalla coda o sospende il processo se la coda è vuota

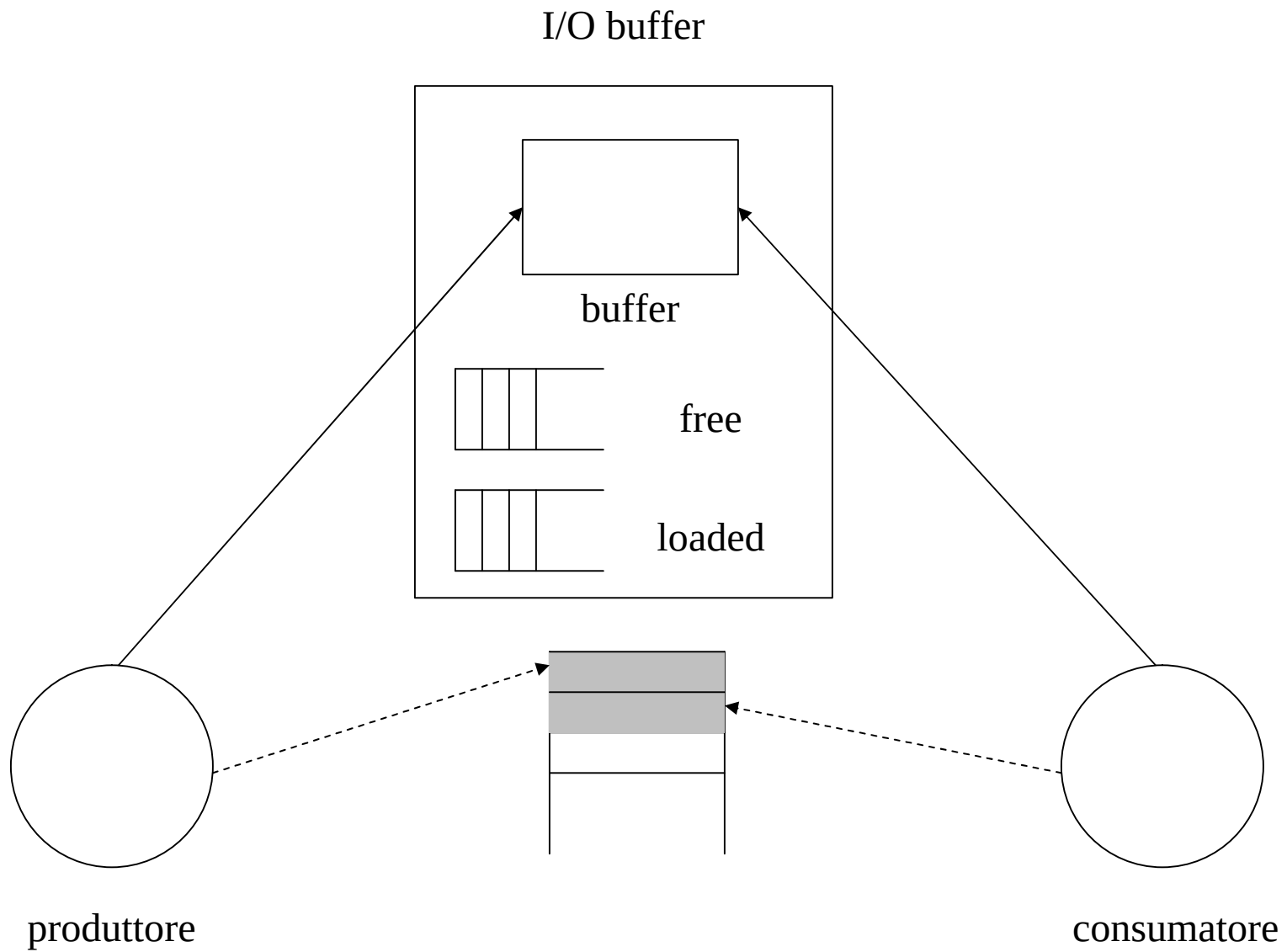
ESEMPIO: I/O BUFFER

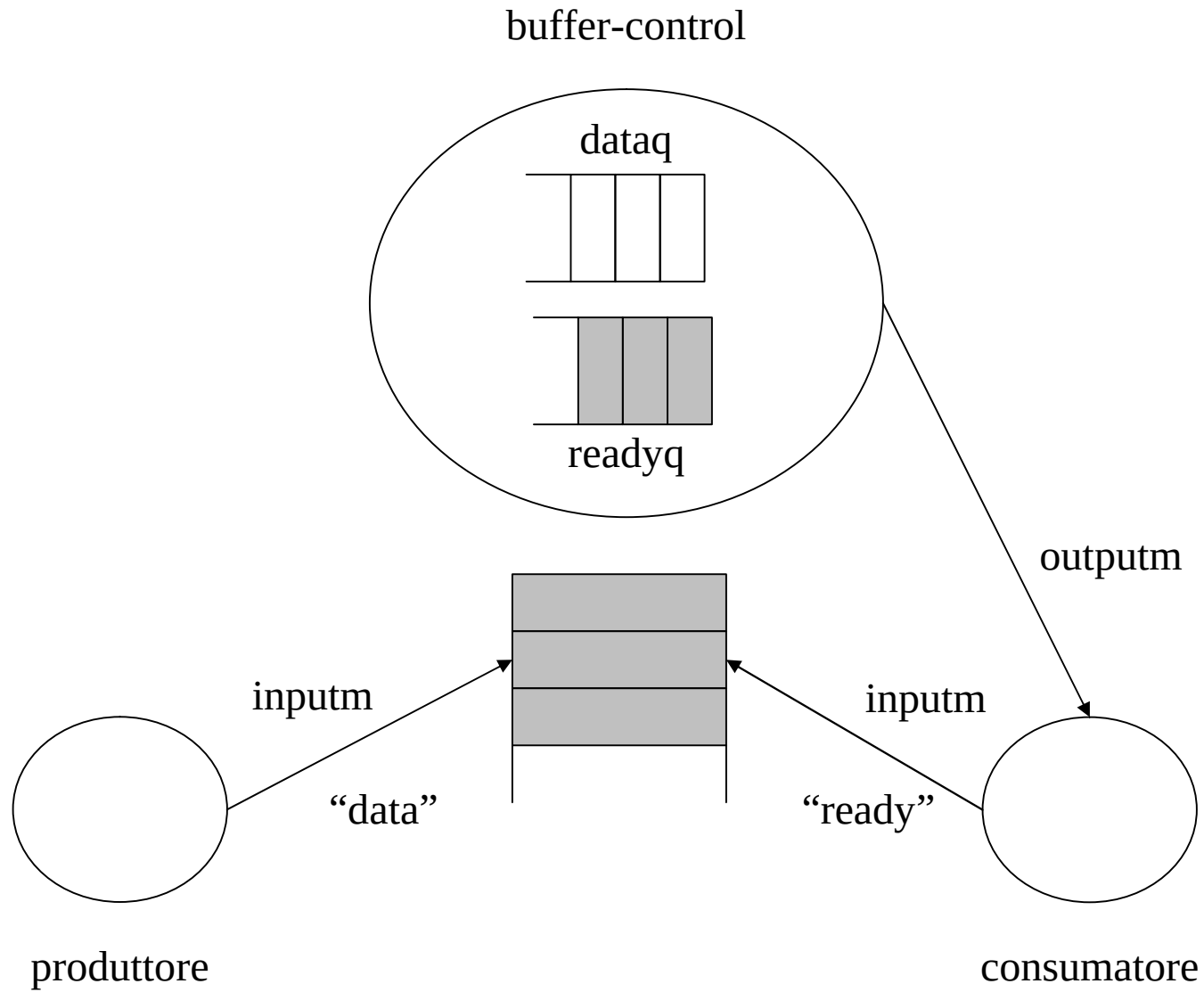


- P_i manda un messaggio al processo *buffer-control* che a sua volta lo invia ad uno dei processi C_j
- Ciascun C_j deve inviare un messaggio a *buffer-control* per indicare che è **pronto** a ricevere il messaggio
- Esistono quindi due tipi di **messaggi di input** a *buffer-control*
 - “**data**” inviato da P_i
 - “**ready**” inviato da C_j
- Devono esistere **due code** entro *buffer-control* per memorizzare i **due tipi di messaggi**: “**dataq**” e “**readyq**”

```
/* soluzione nel modello a memoria comune,  
   mediante il monitor: */
```

```
monitor IObuffer  
{ block buffer;  
  boolean inuse=false;  
  condition free, loaded;  
  
  public void deliver(block in)  
  {   if (inuse) free.wait;  
      buffer= in;  
      inuse=true;  
      loaded.signal;  
  }  
  
  public void retrieve(block *out)  
  {   if (!inuse) loaded.wait;  
      *out=buffer;  
      inuse=false;  
      free.signal;  
  }  
}
```





/ soluzione nel modello a scambio di messaggi:
processo gestore dell'I/O Buffer: */*

```
void buffer-control() /* codice processo gestore */;
{ message inputm, outputm;
  messagequeue dataq , readyq;
  while (1)
  {   receive (&inputm);
      if (inputm.contents.type==data)
          if <ci sono messaggi nella coda readyq>
          {   <prepara outputm>;
              send (outputm);}
          else <inserisci inputm nella coda dataq>;
      else if (inputm.contents.type==ready)
          if <ci sono messaggi nella coda dataq>
          {   <prepara outputm>;
              send (outputm);}
          else <inserisci inputm nella coda readyq>;
      }
  }
```

- **Sincronizzazione tra processi attraverso scambi di messaggi :**
 - Nell'esempio precedente, prima di inviare un messaggio di tipo "data", il processo produttore può inviare un messaggio "ready-to-send" a buffer-control, che risponderà "send" se c'è **posto nella coda**.
- **Riduzione del parallelismo**
 - Non è possibile l'accesso contemporaneo ad una risorsa da parte di più processi. Gli accessi sono sequenzializzati dal processo servitore.
 - Si può ovviare suddividendo la risorsa in più risorse gestite da processi diversi

Costrutti Linguistici per il Modello a Scambio di Messaggi

Classificazione in base a due caratteristiche:

- a) **designazione** dei processi sorgente e destinatario di ogni comunicazione
- b) **tipo di sincronizzazione** tra processi comunicanti

Caratteristiche **ortogonali**: le soluzioni proposte per a) e b) sono tra loro indipendenti.

PRIMITIVE DI SINCRONIZZAZIONE BASATE SULLO SCAMBIO DI MESSAGGI

SEND <expression-list> to <destination-designator>;

RECEIVE <variable-list> to <source-designator>;

- l'esecuzione della **send** valuta "*expression list*" che diventa il messaggio; "*destination-designator*" dà al programmatore il controllo su dove inviare il messaggio
- L'esecuzione della **receive** riceve il messaggio nella lista delle variabili "*variable-list*"; "*source-designator*" dà al programmatore il controllo sull'origine dei messaggi.

Designazione dei processi Sorgente e Destinatario di ogni comunicazione

Canale: collegamento logico tramite il quale due processi comunicano.

- E' compito del **supporto a tempo di esecuzione** di un linguaggio fornire l'astrazione *canale* come meccanismo primitivo per lo scambio di informazioni.
- Il linguaggio deve fornire gli **strumenti linguistici** per specificare sia la sorgente che la destinazione di ogni messaggio.

Comunicazione simmetrica:

- Vengono direttamente usati i nomi dei processi per denotare un canale, sia dal mittente che dal destinatario.

Comunicazione asimmetrica:

- Il processo mittente nomina esplicitamente il destinatario
- il processo destinatario non esprime il nome del processo con il quale vuole comunicare.

CANALE DI COMUNICAZIONE

La coppia

“destination-designator/source-designator”

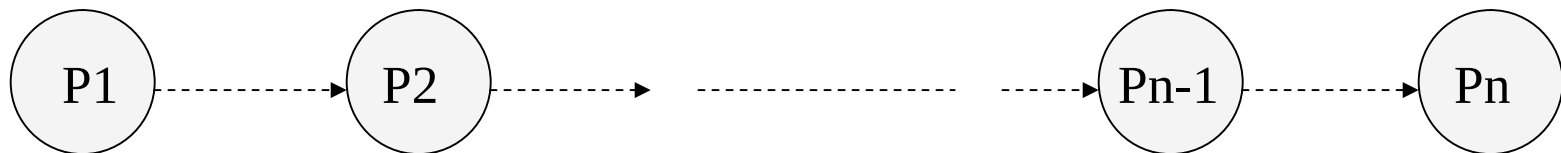
definisce un canale di comunicazione.

Schema simmetrico: i processi si nominano esplicitamente:

send <message> to P2

receive <message> from P1

E' usato nei modelli del tipo “pipeline”.



Esempio: comunicazione simmetrica

```
void reader_process()      /* codice del processo reader*/
{
    cardimage card;
    while(..)
    {
        <read card from cardreader>;
        send card to executer;
    }
}

void executer_process() /* codice del processo executer*/
{
    cardimage card;
    lineimage line;
    while(..)
    {
        receive card from reader;
        <process card and generate line>;
        send line to printer;
    }
}

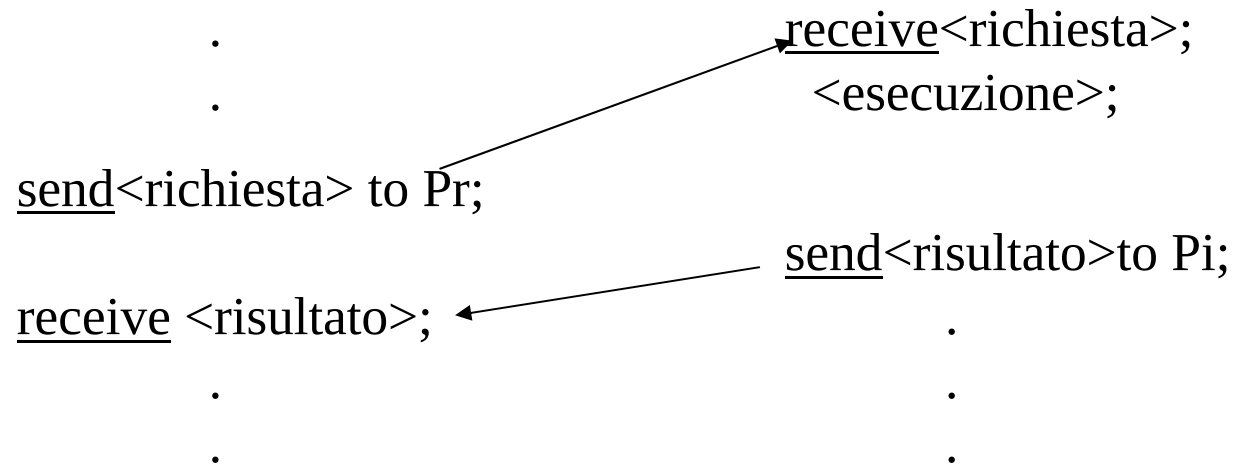
void printer_process() /* codice del processo printer*/
{
    lineimage line;
    while (..)
    {
        receive line from executer;
        <print line on line printer>;
    }
}
```

SCHEMA ASIMMETRICO

- Il mittente nomina esplicitamente il destinatario ma questi, al contrario, non esprime il nome del processo con cui desidera comunicare
- Modello **CLIENTE-SERVITORE** $\Rightarrow$ corrisponde all'uso di un processo come gestore di una risorsa

Pi (cliente)

Pr (servitore)



- **Schema da molti a uno:** i processi clienti specificano il **destinatario** delle loro richieste. Il processo servitore è pronto a ricevere messaggi da **qualunque cliente**.
- **Schema da uno a molti o da molti a molti:** processi cliente che inviano richieste non ad un particolare servitore, ma ad **uno qualunque** scelto tra un insieme di **servitori equivalenti**
- Equivalente al caso della **mailbox** in ambiente a **memoria comune**; non è specificata l'identità del particolare servitore
- Problemi di **natura realizzativa**
- Il supporto a tempo di esecuzione del linguaggio deve garantire che un messaggio di richiesta sia **inviato a tutti i processi servitori** ed assicurare che, non appena il messaggio è ricevuto da uno di essi, lo stesso **non sia più disponibile per tutti gli altri servitori**

Tipi di sincronizzazione dei processi comunicanti

Tre alternative per la semantica della *send*:

- ***send* asincrona**
- ***send* sincrona**
- ***send* tipo *chiamata di procedura remota* (RPC)**

Due alternative per la *receive*:

- ***receive* bloccante**
- ***receive* non bloccante**

send asincrona

- Il processo mittente **continua la sua esecuzione** immediatamente dopo che il messaggio è stato inviato.
- Il messaggio ricevuto contiene informazioni che non possono essere associate allo stato attuale del mittente (difficoltà nella verifica dei programmi).
- La send non è un punto di sincronizzazione. E' necessario pertanto prevedere, per realizzare un particolare schema di interazione, uno scambio esplicito di messaggi che non contengono informazioni da elaborare.

Primitiva di basso livello (paragone con il meccanismo semaforico)

- Facilità di realizzazione in termini del meccanismo primitivo del supporto a tempo di esecuzione.
- Carenza espressiva (difficoltà di uso del meccanismo e di verifica dei programmi).
- **Buffer di messaggi.** Richiede un buffer di *capacità illimitata*. Se si vuole mantenere immutata la semantica della send occorre che il supporto a tempo di esecuzione provveda a sospendere il processo in caso di coda piena (problemi di *deadlock nascosti*)

Modifica della semantica della send:

- Il processo mittente si **blocca** qualora la coda dei messaggi sia piena;
- La primitiva ***send***, in caso di coda piena, solleva **un'eccezione** che viene notificata al processo mittente.

Send sincrona (“rendez-vous” semplice)

- Il processo mittente si blocca in attesa che il messaggio sia stato ricevuto.
- Ogni messaggio ricevuto contiene informazioni corrispondenti **allo stato attuale** del processo mittente.
- Ciò semplifica la scrittura e la verifica dei programmi.

Send tipo *chiamata a procedura remota* (“rendez-vous” esteso)

- Il processo mittente rimane **in attesa** fino a che il ricevente non ha terminato di **svolgere l'azione richiesta**.
- Analogia semantica con la chiamata di procedura.
- Modello cliente-servitore.
- Riduzione di parallelismo (apparente).
- Punto di sincronizzazione: semplificazione della verifica dei programmi.

Receive

Normalmente è **bloccante** se non ci sono messaggi sul canale; costituisce un **punto di sincronizzazione**.

Problema:

un processo desidera ricevere **solo alcuni** messaggi ritardando l'elaborazione di altri (gestori di risorse).

Soluzione:

specificare **più canali** di ingresso per ogni processo, ciascuno dedicato a messaggi di tipo **diverso**.

Deve essere possibile inoltre specificare su quali canali attendere, sulla base dello **stato interno** della risorsa.

In caso di presenza contemporanea di messaggi su questi canali la scelta può essere fatta secondo criteri di priorità e in modo non deterministico.

- Si può ricorrere a una primitiva che **verifica** lo stato di un canale, restituisce un messaggio se presente o un'indicazione di canale **vuoto** (non bloccante)

- Ciò consente ad un processo di selezionare l'insieme di canali da cui prelevare un messaggio.
- Inconveniente: qualora sia necessario attendere un messaggio da uno specifico canale occorre far uso di cicli di attesa attiva.
- Una delle soluzioni più adottate per garantire le stesse funzionalità di una primitiva non bloccante, evitando le **attese attive** e consentendo la **verifica dello stato dei canali** è quella che fa uso dei

comandi con guardia

COMANDI CON GUARDIA

- L'uso della *receive* bloccante può risultare complesso qualora un processo desideri ricevere solo alcuni messaggi ritardando l'elaborazione degli altri (processi **gestori di risorse**: ricezione di messaggi compatibili con lo stato delle risorse)

Comandi con guardia: sono istruzioni del tipo

<guardia> $\longrightarrow$ <istruzione>

dove <guardia> è *un'espressione logica* (guardia logica) seguita da un'eventuale *primitiva receive* (guardia di ingresso).

- La guardia viene valutata quando il comando deve essere eseguito. Si ha:
 - **guardia fallita** se l'espressione logica è **falsa**
 - **guardia valida** se l'espressione logica è **vera** e se *receive* può essere eseguita senza ritardi
 - **guardia ritardata** se l'espressione logica è **vera** ma *receive non può essere eseguita*

Comando con guardia *alternativo*:

```
if B1  —————>      istruzione 1;  
• B2  —————>      istruzione 2;  
.  
.  
• Bn  —————>      istruzione n;  
end;
```

tra tutte le B_i valide ne viene scelta una in modo **non deterministico**, viene eseguita **receive** e quindi l'istruzione associata.

–Se tutte le guardie **falliscono** il comando viene **abortito**

–Se tutte le guardie non fallite sono ritardate il processo si blocca in attesa che almeno una diventi valida

Comando con guardia *ripetitivo*:

do

- B1 → istruzione1;
- .
- .
- Bn → istruzione n;

end

Il comando viene ripetuto finché tutte le guardie falliscono. In questo caso il comando termina la sua esecuzione.

- L'uso dei comandi con guardia consente ad un processo di rimanere in attesa **sull'insieme di canali** per i quali è verificata la corrispondente guardia logica e di **prelevare**, non appena pronto, **il primo dei messaggi in arrivo**.
- Ciascuna guardia logica contraddistingue uno **stato della risorsa compatibile** con la ricezione di questi messaggi.