MODELLI DI INTERAZIONE TRA PROCESSI

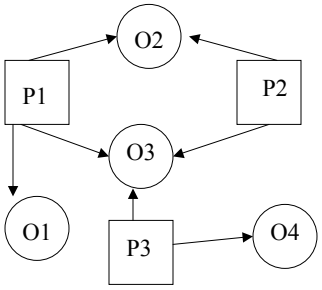
- **Modello ad ambiente globale ("global environment")**
- **Modello a scambio di messaggi ("message passing")**

MODELLO AD AMBIENTE GLOBALE

Il sistema è visto come un insieme di

- processi

- oggetti (risorse)



Diritto di accesso

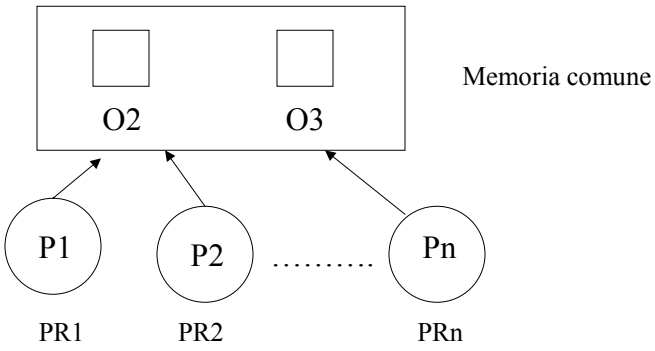
O1 , O4 risorse private

O2 , O3 risorse comuni

Tipi di interazioni tra processi

- competizione
- cooperazione

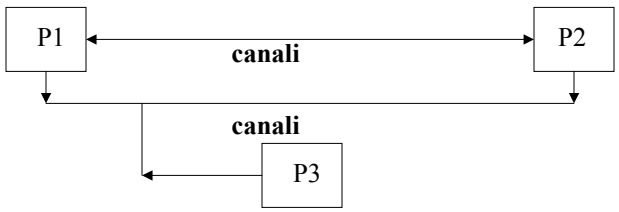
- Il modello ad ambiente globale rappresenta la naturale astrazione del funzionamento di **un sistema in multiprogrammazione** costituito da uno o più processori che hanno accesso ad una memoria comune



- Ad ogni processore può esasere associata una memoria privata, ma ogni interazione avviene tramite oggetti contenuti nella memoria comune

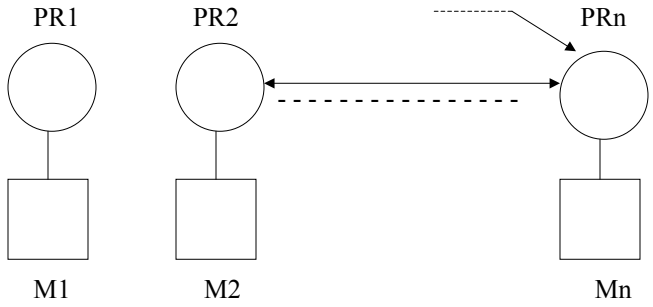
MODELLO A SCAMBIO DI MESSAGGI

Il sistema è visto come un insieme di processi ciascuno operante in un ambiente locale che non è accessibile a nessun altro processo



- Ogni forma di interazione tra processi (comunicazione, sincronizzazione) avviene tramite scambio di messaggi
- Non esiste più il concetto di **risorsa accessibile direttamente ai processi;** sono possibili due casi:
  - alla risorsa è associato un **processo servitore**
  - la risorsa viene passata da un processo all'altro **sotto forma di messaggi**

- Il modello a scambio di messaggi rappresenta la naturale astrazione di un sistema privo di memoria comune, in cui a ciascun processore è associata una memoria privata



- Il modello a scambio di messaggi può essere realizzato anche in presenza di memoria comune, che viene utilizzata per realizzare dei canali di comunicazioni

PRIMITIVE PER LO SCAMBIO DI MESSAGGI

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

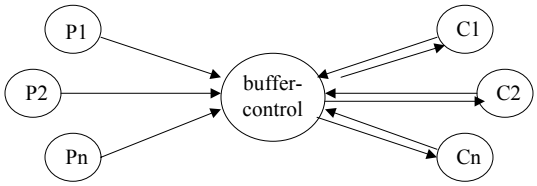
```
type messaggio = record
                   origine: …. ;
                   destinazione: …. ;
                   contenuto: …. ;
                   end
```

- 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 **var m: message**

- 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



- Pi manda un messaggio al processo *buffer-control* che a sua volta lo invia ad uno dei processi Cj
- Ciascun Cj 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 Pi
  - **"ready"** inviato da Cj
- Devono esistere **due code** entro *buffer-control* per memorizzare **i due tipi di messaggi**

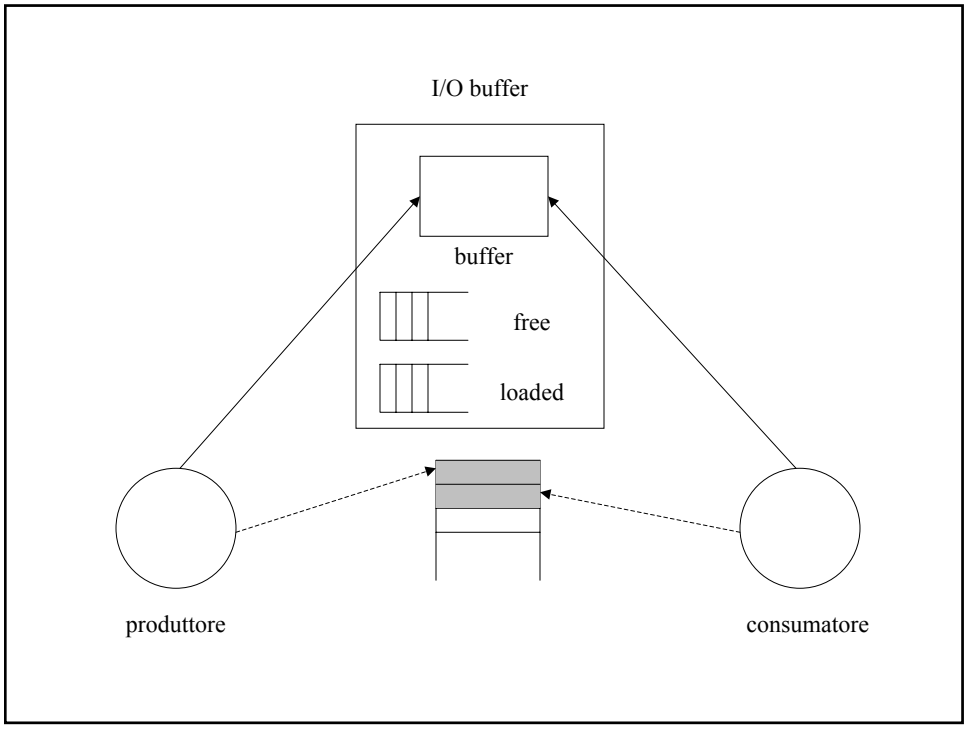
"dataq" e **"readyq"**

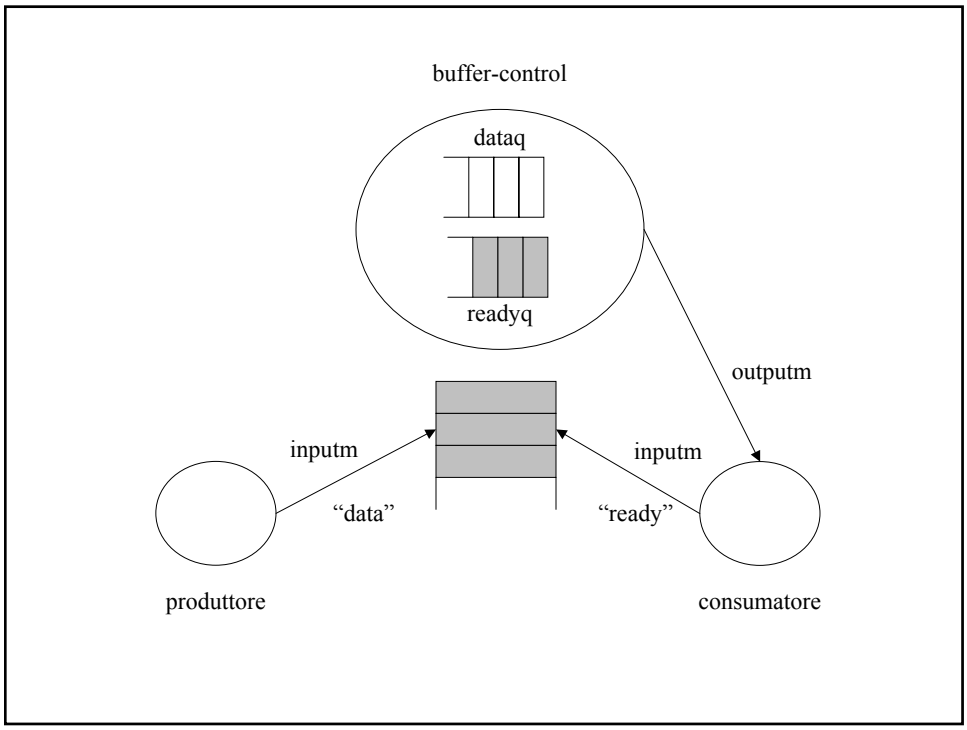
```
type I/O buffer = monitor;
var buffer : block; inuse : boolean; free, loaded : condition;

procedure entry deliver (in : block);
   begin
      if inuse then free.wait;
      buffer : = in;
      inuse := true
      loaded.signal
   end;

procedure entry retrieve (out : block);
   begin
      if not inuse then loaded.wait;
      out := buffer;
      inuse := false
      free.signal;
   end

begin inuse := false end
end
```


I/O buffer
buffer
free
loaded
produttore
consumatore


buffer-control
dataq
readyq
outputm
inputm
"data"
inputm
"ready"
produttore
consumatore

```
process buffer-control;
var inputm, outputm : message;
      dataq , readyq : queue of messages;

cycle
   receive (inputm);
   case inputm.contents.type of
      "data": if <ci sono messaggi nella coda readyq>
                then <prepara outputm>;
                     send (outputm)
                else <inserisci inputm nella coda dataq>
      "ready": if <ci sono messaggi nella coda dataq>
                 then <prepara outputm>;
                      send (outputm)
                 else <inserisci inputm nella coda readyq>
end
```

- **Sicronizzazione 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 sequenzializzatti 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:

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;

  "source-designator" dà al programmatore il controllo sull'origine dei messaggi

Destinazione dei processi sorgente e destinatario di ogni comunicazzione

- *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

Comunicazione asimmetrica

Il processo mittente nomina esplicitamente il destinatario, ma questi 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

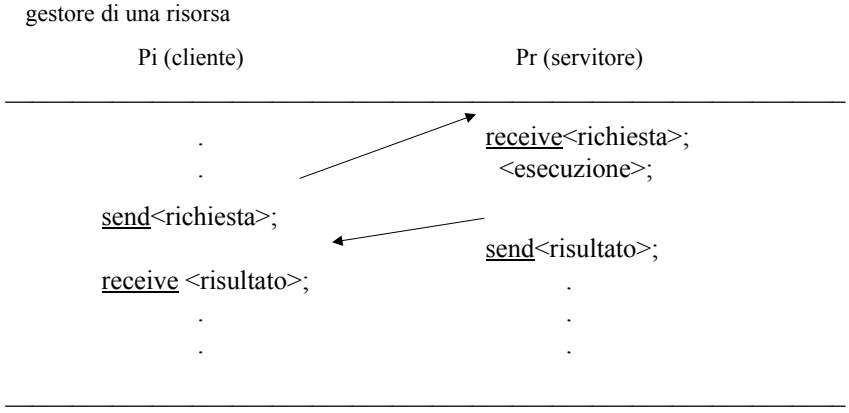
receive <message> from P1

- E' usato nei modelli del tipo "pipeline"


```
process reader;
   var card: cardimage;
       loop
            <read card from cardreader>;
            send card to executer;
       end;
process execute;
   var card: cardimage;
        line: lineimage
       loop
            receive card from reader;
            <process card and generate line>;
            send line to printer;
       end;
process printer;
   var line: lineimage;
       loop
            receive line from executer;
            <print line on line printer>;
       end;
```

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 ⇨ corrisponde all'uso di un processo come gestore di una risorsa



- Schema da **molti a uno**: i processi clienti specificano il **destinatario** delle loro richieste. I l processo servitore è pronto a ricevere messaggi da **qualunque cliente.**

- Shema da **uno a molti** o da **molti a molti**: processi cliente che inviano richieste non ad un parrticolare 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 diponibile per tutti gli altri servitori**

Tipo di sincronizzazione dei processi comunicanti

Tre alternative per la semantica della *send*:

send asincrona

send sincrona

send tipo *chiamata di procedura remota*

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.
Un 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’eleborazione 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 guardie.**

<guardia> → <istruzione>

<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** " " " 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 Bi 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 esecizione.

- 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.