Primitive asincrone

- **Send** *non bloccante*: il processo mittente, non appena inviato il messaggio, *prosegue* la sua esecuzione.
- Il supporto a tempo di esecuzione deve fornire un meccanismo di accodamento dei messaggi.
- Il processo ricevente non può dedurre dal contenuto del messaggio ricevuto *alcuna informazione* sullo stato del mittente.

Esempio produttori consumatori

- Utilizzo delle primitive *send e receive asincrone* per realizzare uno schema cliente-servitore.
- Il servitore è un processo che gestisce una *risorsa buffer* destinata a memorizzare i messaggi inviati da uno o più processi produttori ad uno o più processi consumatori.
- Ogni processo produttore può inviare un messaggio ad uno *qualunque* dei processi consumatori.*Si vogliono evitare situazioni in cui esistano processi consumatori in attesa di messaggi ed altri con messaggi in coda.*

- Il processo *buffer* gestisce la risorsa di memorizzazione dei messaggi (*dati*) inviati dai processi produttori e convoglia tali messaggi in ordine *FIFO* ai processi consumatori.
- Ogni consumatore invia un messaggio di controllo (*pronto*) al *buffer* per denotare il suo stato di pronto a ricevere un messaggio e quindi si mette in *attesa* del messaggio.
- *Buffer* è un processo *servitore*, produttori e consumatori sono processi *clienti*.

Primitive:

```
Send(mes, proc);
proc=Receive(&mes);
```

- ***mes*** è una variabile di tipo messaggio, ***proc*** è una variabile di tipo *process*
- Ogni processo ha una *coda di messaggi* a lui inviati gestita *FIFO*.
- La primitiva **Send** inserisce il messaggio ***mes*** nella coda di ***proc*** e termina
- La primitiva **Receive** analizza la coda d'ingresso del processo che la esegue: se la coda è vuota il processo viene bloccato; diversamente viene estratto il primo messaggio il cui valore è assegnato alla variabile ***mes***.

Due diversi tipi di messaggi per rappresentare le diverse informazioni scambiate tra i processi:

- i messaggi di tipo *in-mess* ricevuti dal processo buffer : sono caratterizzati da due varianti per tenere conto sia dei ***dati*** inviati dai produttori che dei *segnali di controllo* (***pronto***) inviati dai consumatori.
- i messaggi di tipo *out-mess* inviati da *buffer* ai consumatori (***dati***).

- I messaggi inviati a *buffer* sono ricevuti in ordine *FIFO* (*dati* o segnali *pronto*).
- Quando viene ricevuto un messaggio inviato da un produttore (*dati*):
  - questo deve essere inviato ad un consumatore, se uno o più consumatori sono in attesa;
  - diversamente il messaggio deve essere memorizzato nella *coda-dati* locale a *buffer*.
- Quando viene ricevuto un messaggio da un consumatore (segnale *pronto*):
  - se nella coda locale vi sono dati disponibili, ne viene prelevato uno ed inviato al consumatore;
  - in caso contrario il nome del consumatore viene inserito nella *coda-consumatori–pronti* contenente i nomi dei consumatori in attesa di dati.

```
typedef enum{dati,pronto}msg_ricevuto;

typedef struct{
  msg_ricevuto specie;
  union{
     tipodato informazione; /* specie == dati*/
     tiposegnale ready; /* specie == pronto*/
     }contenuto;
  }in-mess;

typedef tipodato out-mess;
```

```
void Process_buffer() /* codice del processo buffer*/
{  T_queue coda_dati; /*coda di elementi di tipo Tipodato*/
   P_queue consumatori_pronti; /*coda di nomi di processo*/
   Tipodato inf;
   process consumatore, proc;
   in-mess in; out-mess out;
   while (1)
   {    proc= Receive(&in);
        switch (in.specie)
        {case dati:
              if (<consumatori-pronti e` vuota>)
                    <inserzione di
              in.contenuto.informazione
                    nella  coda_dati>;
              else
              {     <estrazione di consumatore da
              consumatori-pronti>;
                    out=in.contenuto.informazione:
                    Send(out,consumatore);
              }
              break;
        case pronto: if (<coda_dati e` vuota>)
                          <inserzione proc in
                          consumatori-pronti>;
                    else { <estrazione di inf da coda_dati>;
                          out=inf;
                          Send(out,proc);}
        }/* fine switch*/
   }/* fine while*/
}
```

Codice produttore:

```
void produttore-i()
{  in-mess mess;
   Tipodato C;
   mess.specie=dati;
   while(1)
   {  <produci contenuto C>;
      mess.contenuto.informazione=C;
      Send(mess,buffer);
   }
}
```

Codice consumatore:

```
void consumatore-j()
{  in-mess mess1, mess2;
   mess1.specie=pronto;
   while(1)
   {  Send(mess1,buffer);
      proc=Receive(&mess2);
      <consuma mess2>;
   }
}
```

Uso delle porte

- Più *canali di ingresso* per ogni processo ciascuno dedicato a messaggi di *tipo diverso*.
- Utilizzo di uno schema di ricezione che consenta di specificare, sulla base dello *stato interno* del processo ricevente, l'insieme dei canali sui quali attendere il messaggio.

Definizione di una *porta x*:

```
port x:T; /* variabile di tipo porta*/
```

- dove *T* e` il tipo dei messaggi associati alla porta.
- L'identificatore della porta è visibile all'*esterno* del processo.
- Nel caso di più porte con lo stesso nome, si utilizza la notazione:

nome-processo.nome-porta

Primitive:

```
Send_to(x, m);
proc= Receive_from(x,&m);
```

- Due porte per il processo *buffer*, una per ricevere i messaggi *dati* e l'altra per messaggi *pronto.*
- Il tipo della prima porta corrisponde al tipo *T* dei dati inviati dai produttori.
- Il tipo della seconda porta corrisponde al tipo predefinito *signal* che contraddistingue i messaggi contenenti solo segnali di *sincronizzazione.*
- Il supporto a tempo di esecuzione deve fornire *tante code di ingresso* per ciascun processo quante sono *le porte* in esso dichiarate

```
void Process_buffer()
{ port porta-dati:T;
  porta controllo:signal;
  T inf;
  signal pronto;
  process produttore,consumatore;
  do  /* comando con guardia ripetitivo*/
  □ produttore=Receive_from(porta-dati, &inf);
      → consumatore=Receive_from(controllo, &pronto);
      Send_to(consumatore.inport, Inf);
  □ consumatore= Receive_from(controllo,&pronto);
      → produttore:=Receive_from (porta-dati,&inf);
      Send_to(consumatore.inport, Inf);
  end;
}
```

Codice produttore:

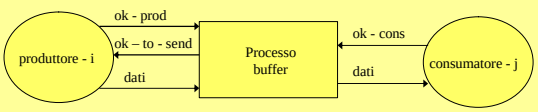
```
void produttore-i()
{     T inf;
      while (1)
      {     <produci contenuto>;
            inf=<contenuto>;
            Send_to(porta-dati, inf);
      }
}
```

Codice consumatore:

```
void consumatore-j()
{     T inf;
      process buffer;
      signal pronto;
      port inport:T;
      while(1)
      {     Send_to(controllo, pronto);
            buffer=Receive_from(inport,inf);
            <consuma inf>;
      }
}
```

Esempio produttori-consumatori con limitazione del numero di messaggi memorizzati

- N numero di messaggi memorizzati da *buffer.*
- Il produttore rimane in attesa di un segnale di abilitazione all'invio dei dati da parte di *buffer* e soltanto quando tale messaggio è arrivato invia il messaggio *dati.*




```
void Process_buffer()
{ port porta-dati:T;
  port controllo-prod: signal;
  port controllo-cons: signal;
  T inf; signal ok-prod, ok-cons, ok-to-send;
  process prod, cons;
  int cont=0; /* contatore dei messaggi nel buffer*/
  do
  □(cont<N); prod=Receive_from(controllo-prod,&ok-prod);
      → cont++;
      Send_to(prod.okport, ok-to-send);
  □(cont>0); cons=Receive_from(controllo-cons,&ok-cons);
      → prod=Receive_from(porta-dati,&inf);
      cont--;
      Send_to(cons.inport, inf);
  end;
}
```

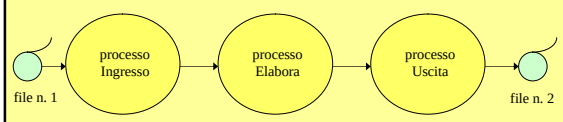
```
void produttore_i()
{ port okport: signal;
  T inf;
  signal ok-prod, ok-to-send;
  process buffer;

  while (1)
  {   <produci contenuto>;
      inf=<contenuto>;
      Send_to(controllo-prod, ok-prod);
      buffer=Receive_from(okport, ok-to-send);
      Send_to(porta-dati,inf);
  }
}
```

- Lo schema del processo consumatore rimane lo stesso.

ESEMPIO DI SISTEMA PIPELINE

- Lettura degli N record di un file, loro elaborazione e creazione di un secondo file contenente gli N record elaborati.
- Il processo di ingresso legge da file_1 un record alla volta e lo invia al processo di elaborazione il quale, a sua volta, riceve il record, lo elabora e lo invia al processo di uscita.
- Quest'ultimo riceve, uno alla volta, i record elaborati e li scrive sul file_2.




```
/* Programma esempio-pipeline */

FILE file1,file2; /*file of T*/
process Ingresso, Elabora, Uscita;
...
/* codice del proc.
Ingresso: */
void Proc_Ingresso()
{T buf;
 int i;
 for(i=1; i<=N; i++)
 { Read(file1,&buf);
   Send(buf, Elabora);
 }
}
```

```
/* codice del proc. Elabora: */
void Proc_Elabora()
{T buf-in, buf-out;
 process proc; int i;
 for(i=1;i<=N; i++)
 {          proc=Receive(&buf-in);
            buf-out:=<risultato elaborazione
buf-in >;
            Send(buf-out,Uscita);
 }
}
```

```
/* codice del proc. Uscita: */
void Proc_Uscita()
{T buf;  process proc;
 int i;
 for(i=1;i<=N; i++)
 {          proc=Receive(&buf);
            write(file2,buf);
 }
}
```

- L'uso di primitive con specifiche di canale di tipo simmetrico potrebbe risultare più indicato in questo caso.
- Un controllo del flusso dei messaggi può essere ottenuto per ogni copia di processi con l'utilizzazione di un processo buffer.