Problema dei Produttori e dei Consumatori

- Appartiene alla categoria di problemi nei quali l'esecuzione di un processo dipende dall'esecuzione di un altro, per cui è necessaria una comunicazione asincrona tra i due processi
- Due categorie di processi
  - Produttori, che scrivono su di una risorsa condivisa
  - Consumatori, che leggono e azzerano la risorsa condivisa

Vincoli del problema

1) Il produttore non può produrre un messaggio prima che qualche consumatore abbia letto il messaggio precedente
2) il consumatore non può prelevare alcun messaggio fino a che un produttore non l'abbia depositato

Nel caso più generale i messaggi sono depositati in un buffer di lunghezza N:

1') Il produttore non può produrre un messaggio se il buffer è pieno
2') Il consumatore non può prelevare messaggi se il buffer è vuoto

Una semplice soluzione

- Il buffer è una risorsa condivisa tra i processi (ad esempio un segmento di memoria condivisa), gestita come una coda
- L'accesso dei produttori è regolato dal semaforo N-ario *spazio_disponibile*
- L'accesso dei consumatori è regolato dal semaforo N-ario *messaggio_disponibile*

```
semaforo messaggio_disponibile=0:
semaforo spazio_disponibile = N;
queue_message coda;

void invio (messaggio m)
{   wait (spazio_disponibile);
        accoda il messaggio m
    signal (messaggio_disponibile); }

messaggio ricezione()
{   messaggio m;
    wait (messaggio_disponibile);
        m= preleva primo messaggio in
    coda
    signal (spazio_disponibile);
    return m; }
```

```
semaforo messaggio_disponibile=0:
semaforo spazio_disponibile = N;
semaforo mutex_prod=1;
semaforo mutex_cons=1;
messaggio buffer[N];
int testa, coda;

void invio (messaggio m)
{  wait (spazio_disponibile);
       wait(mutex_prod);
            buffer[coda]=m;
            coda=(coda++)mod N;
       signal(mutex_prod);
   signal (messaggio_disponibile);
}
messaggio ricezione()
{
   messaggio m;
   wait (messaggio_disponibile);
       wait(mutex_cons);
       m= buffer[testa];
       testa= (testa++) mod N;
       signal(mutex_cons);
   signal (spazio_disponibile);
   return m;
}
```

Problemi di questa soluzione

- Ogni processo impegna il buffer in tutta la durata della produzione/consumazione

$\Rightarrow$ non è possibile alcun parallelismo tra i produttori (consumatori)

$\Rightarrow$ non è possibile distinguere tra richiesta, uso e rilascio della risorsa buffer

Una seconda soluzione

- Due vettori:
  - buffer [N], che contiene i valori
  - stato [N], nel quale ogni cella può contenere tre possibili valori:
    - PIENO, se la cella corrispondente nel buffer contiene un valore prodotto;
    - VUOTO, se la cella corrispondente non contiene un valore consumabile;
    - IN_USO, se la cella corrispondente è oggetto di un processo produttore o consumatore attualmente attivo

Una seconda soluzione

- Scomposizione del problema in 6 funzioni:
  - Richiesta, Produzione e Rilascio per i produttori;
  - Richiesta, Consumo e Rilascio per i consumatori.
- Ipotesi semplificativa:
  - ogni consumatore è disposto a leggere un qualsiasi messaggio, tra quelli presenti nel buffer

Strutture dati

```
# define VUOTO 0
# define PIENO 1
# define IN_USO 2
semaforo messaggio_disponibile=0:
semaforo spazio_disponibile=N;
semaforo mutex_prod=1;
semaforo mutex_cons=1;
messaggio buffer[N];
int stato[N];
/* buffer e stato sono in memoria
  condivisa */
typedef char messaggio
```

Produttore

```
int Richiesta_Produttore() {
  int indice;
  wait(spazio_disponibile);
     wait(mutex_prod);
     indice=0;
     while (stato[indice]!=VUOTO)
          indice++;
     stato[indice]=IN_USO;
     signal(mutex_prod);
  return indice; }

void Produzione (int indice,
  messaggio valore)
{ buffer[indice]=valore;}

void Rilascio_Produttore (int
  indice) {
  stato[indice]=PIENO;
  signal (messaggio_disponibile);
}
```

Consumatore

```
int Richiesta_Consumatore() {
  int indice;
  wait(messaggio_disponibile);
     wait(mutex_cons);
     indice=0;
     while (stato[indice]!=PIENO)
          indice++;
     stato[indice]=IN_USO;
     signal(mutex_cons);
  return indice; }

messaggio Consumo (int indice)
{ return buffer[indice];}

void Rilascio (int indice) {
  stato[indice]=VUOTO;
  signal (spazio_disponibile); }
```

Commenti

- Con questa soluzione può esistere un parziale parallelismo tra i produttori: possono coesistere un produttore che richiede una risorsa e altri che producono o rilasciano risorse
- Lo stesso vale in maniera analoga per i consumatori

Implementazione: Costanti e prototipi

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

typedef char messaggio;

#define VUOTO  0
#define PIENO  1
#define IN_USO 2

#define PROD 0
#define CONS 1

void wait_sem(int id_sem, int num_sem);
void signal_sem(int id_sem, int num_sem);

int Richiesta_produttore(void);
void Produzione(int indice, messaggio valore);
void Rilascio_produttore(int indice);

int Richiesta_consumatore(void);
messaggio Consumo(int indice);
void Rilascio_consumatore(int indice);
```

Inizializzazione semafori e memoria condivisa

```
main()
…  dichiarazione variabili …

id_buffer = shmget(chiave, N * sizeof(char),
   IPC_CREAT | 0644);
buffer = shmat(id_buffer, 0, 0);

id_stato =  shmget(chiave, N * sizeof(short
   int), IPC_CREAT | 0644);
stato = shmat(id_stato, 0, 0);

sem = semget(IPC_PRIVATE, 2, IPC_CREAT |
   0644);
semctl(sem, PROD, SETVAL, N);
semctl(sem, CONS, SETVAL, 0);

mutex = semget(IPC_PRIVATE, 2, IPC_CREAT |
   0644);
semctl(mutex, PROD, SETVAL, 1);
semctl(mutex, CONS, SETVAL, 1);

…
```

Primitive semaforiche

```
void wait_sem(int id_sem, int num_sem)
{
   struct sembuf v_op;

   v_op.sem_num = num_sem;
   v_op.sem_op = -1;
   v_op.sem_flg = SEM_UNDO;

   semop(id_sem, &v_op, 1);
}

void signal_sem(int id_sem, int num_sem)
{
   struct sembuf v_op;

   v_op.sem_num = num_sem;
   v_op.sem_op = 1;
   v_op.sem_flg = SEM_UNDO;

   semop(id_sem, &v_op, 1);
}
```

Funzioni per il Produttore

```
int Richiesta_produttore(void)
{
    int indice;

    wait_sem(sem, PROD);
    wait_sem(mutex, PROD);

    indice = 0;
    while (stato[indice] != VUOTO)
       indice++;
    stato[indice] = IN_USO;
    signal_sem(mutex, PROD);
    return indice;
}

void Produzione(int indice, messaggio
   valore)
{  buffer[indice] = valore; }

void Rilascio_produttore(int indice)
{
   stato[indice] = PIENO;
   signal_sem(sem, CONS);
}
```

Funzioni per il consumatore

```
int Richiesta_consumatore(void)
{
     int indice;

    wait_sem(sem, CONS);
    wait_sem(mutex, CONS);

    indice = 0;
     while (stato[indice] != PIENO)
        indice++;
    stato[indice] = IN_USO;
    signal_sem(mutex, CONS);
     return indice;
}

messaggio Consumo(int indice)
{  return buffer[indice]; }

void Rilascio_consumatore(int indice)
{
    stato[indice] = VUOTO;
   signal_sem(sem, PROD);
}
```

Processi produttore e consumatore

```
void Codice_Produttore(void)
{
 int k, indice; messaggio m='a';
 for (k = 0; k < NUM_PRODUZIONI; k++)
 {
   indice = Richiesta_produttore();
   Produzione(indice, m);
   Rilascio_produttore (indice);
   m++;
   sleep(DELAY_PROD);
 }
}

void Codice_Consumatore(void)
{
 int k, indice; messaggio m;
 for (k = 0; k < NUM_CONSUMI; k++)
 {
   indice = Richiesta_consumatore();
   m = Consumo(indice);
   Rilascio_consumatore (indice);
   /* stampa m */
   sleep(DELAY_CONS);
 }
}
```

Codice processo padre

```
…
for (h = 0; h < NUM_PRODUTTORI; h++) {
   pid = fork();
   if (!pid) {
        Codice_Produttore();
        exit(0);
   }
}
for (h = 0; h < NUM_CONSUMATORI; h++) {
   pid = fork();
   if (!pid) {
        Codice_Consumatore();
        exit(0);
   }
}
for (h = 0; h <
   NUM_PRODUTTORI+NUM_CONSUMATORI; h++)
        pid = wait(&status);
semctl(sem, IPC_RMID, 0);
semctl(mutex, IPC_RMID, 0);
shmctl(id_buffer, IPC_RMID, 0);
shmctl(id_stato, IPC_RMID, 0);
}
```

Una variazione

- Con questa implementazione, ogni consumatore cerca di consumare il primo valore nel buffer partendo da sinistra

$\Rightarrow$Se vogliamo, viceversa, che venga consumato l'elemento prodotto da più tempo, dovremo gestire anche testa (per i consumatori) e coda (per i produttori) del buffer

$\Rightarrow$ Testa e coda dovranno essere in memoria condivisa, ma saranno accedute unicamente nelle fasi di richiesta

$\Rightarrow$ per garantire la mutua esclusione nell'accesso a testa e coda non saranno necessari ulteriori semafori