CONCORRENZA: Mutua Esclusione e Sincronizzazione

Concorrenza: genesi

- Multiprogrammazione:
  - gestione di più processi su un singolo processore.
- Multiprocessing:
  - gestione di più processi su più processori.
- Processi distribuiti
  - cluster

- Competizione tra processi (threads) per ottenere (e condividere) le risorse:
  - Cpu
  - Memoria
  - Canali di I/O
  - Files
  - ecc..

Concorrenza: terminologia

- **Sessione Critica**: porzione di codice all'interno di un processo (thread) che richiede accesso a risorse condivise.
- **Deadlock**: situazione nella quale due o più processi sono impossibilitati dal procedere poiché sono in attesa l'uno dell'altro
- **Livelock**: situazione nella quale due o più processi cambiano continuamente il proprio stato a causa del cambiamento di stato degli altri (senza fare alcun lavoro utile)
- **Mutua esclusione**: requisito per il quale, quando un processo è nella propria sezione critica, nessun altro processo può essere nella propria sezione critica se questa fa riferimento a risorse condivise con il primo processo.
- **Race condition**: situazione nella quale thread o processi leggono e scrivono un dato condiviso e il risultato dipende dalla loro velocità reciproca
- **Starvation**: situazione nella quale un processo non riceve mai l'utilizzo di una risorsa e viene costantemente scavalcato da altri processi

Vantaggi e Problemi derivanti dalla concorrenza

VANTAGGI

benefici sulla esecuzione nonostante il sovraccarico (context switching).
Migliore utilizzazione delle risorse.

PROBLEMI

•*Singolo Processore*

- Condivisione pericolosa: ordine delle operazioni di lettura e scrittura su aree di memoria condivise
- Difficoltà nell'assegnare le risorse ai processi in maniera ottimale
- Difficoltà nella rilevazione degli errori nel codice e dei conflitti di interlacciamento

```
void echo()
{
    char in,out;
    scanf("%c", &in);
    out = in;
    printf("%c", out);
}
```

Condivisione della procedura echo():
- Risparmio dello spazio di memoria
- Due processi concorrenti.
- P1 viene interrotto dopo la scanf,
- P2 esegue tutto echo,
- P1 viene riattivato da scanf in poi e ha perso il dato che aveva letto

Soluzione: un solo processo alla volta (MUTUA ESCLUSIONE)

Vantaggi e Problemi derivanti dalla concorrenza

- *SMP*
  - stessi problemi di un calcolatore a singolo processore (una interruzione può fermare l'esecuzione di un processo in un qualsiasi istante)
  - interlacciamento esecuzione processi paralleli

Processo P1 – Processore1

```
……
   scanf("%c", &in);
……
   out = in;
   printf("&c", out);
……
```

Processo P2 – Processore2

```
……
……
   scanf("%c", &in);
   out = in;
   ……
   printf("&c", out);
```

Il carattere letto da P1 è perso prima di poter essere stampato (perdita di aggiornamento)

Soluzione:* *MUTUA ESCLUSIONE

Un solo programma per volta può entrare nella propria sezione critica

Es.: Un solo programma per volta può inviare comandi alla stampante

Requisiti per la mutua esclusione

I meccanismi che provvedono alla mutua esclusione devono garantire i seguenti requisiti:

1. Un solo processo alla volta deve accedere alla sezione (o risorsa) critica;

2. Un processo fuori della sezione critica non deve interferire con il processo nella sezione critica

3. Ogni processo deve poter accedere dopo un tempo finito di attesa in coda alla risorsa critica (no stallo o starvation)

4. Se nessun processo è nella sezione critica, un processo deve poter entrare nella sezione critica senza attese

5. Non ci devono essere supposizioni sulla velocità di esecuzione relativa dei processi

6. Il tempo di permanenza nella sezione critica è (de)finito

Meccanismi di Mutua Esclusione

Approcci software: i processi, senza ausilio del Sistema Operativo o del linguaggio di programmazione, devono coordinarsi tra loro (Dekker)

- Aumento del tempo di esecuzione
- Errori frequenti

Utilizzo di particolari **istruzioni di macchina** (test-set, scambio)

- Riduzione del sovraccarico
- Non soddisfacenti

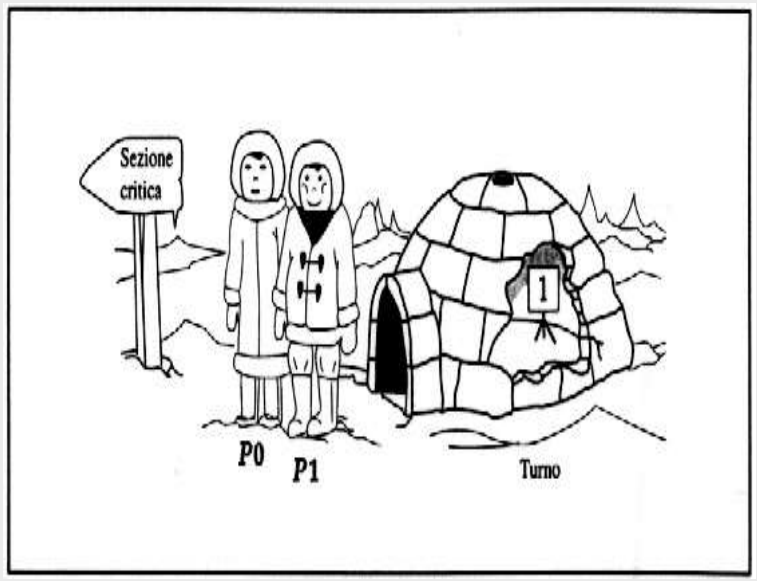
Supporto del sistema operativo o del linguaggio di programmazione

- Scambio Messaggi (Inter Process Communication)
- Semafori
- Monitor

Mutua Esclusione: un approccio software

Algoritmo di Dekker
1° tentativo

MUTUA ESCLUSIONE: Un solo accesso per volta accede alla risorsa condivisa



Protocollo dell'Iglù: prima di entrare nella sezione critica, i processi controllano uno alla volta una variabile turno

Busy wait: consuma tempo utile di esecuzione

```
var turno= 0..1; //variabile globale condivisa
```

Processo 0

```
    ….
while (turno!=0)
{nulla} //busy wait
<sezione critica>;
turno=1;
    …
```

Processo 1

```
….
while (turno!=1)
{nulla} //busy wait
<sezione critica>;
turno=0;
….
```

PRO: garantisce la mutua esclusione

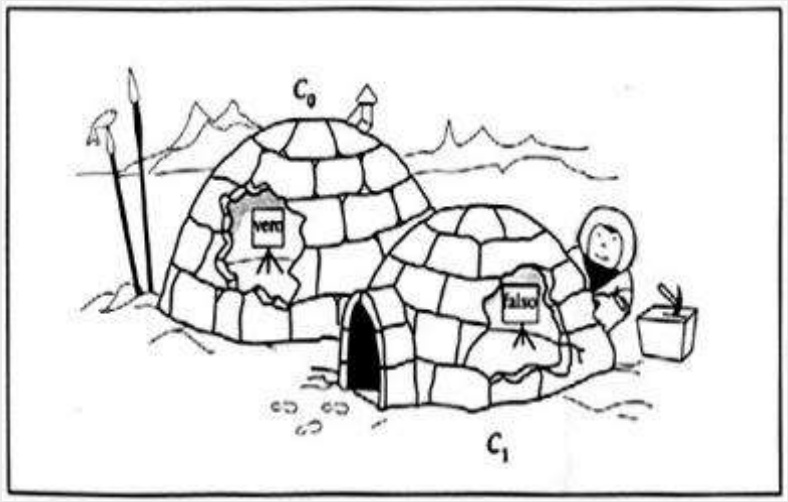
PUNTI DEBOLI:

1.I processi devono osservare l'alternanza, il più lento determina la velocità di avanzamento di entrambi i processi

2. Se un processo fallisce nella propria sezione critica, l'altro processo rimarrà bloccato per sempre

Algoritmo di Dekker
2° tentativo

Ogni processo ha un flag relativo all'utilizzo della risorsa e può leggere il flag dell'altro senza modificarlo



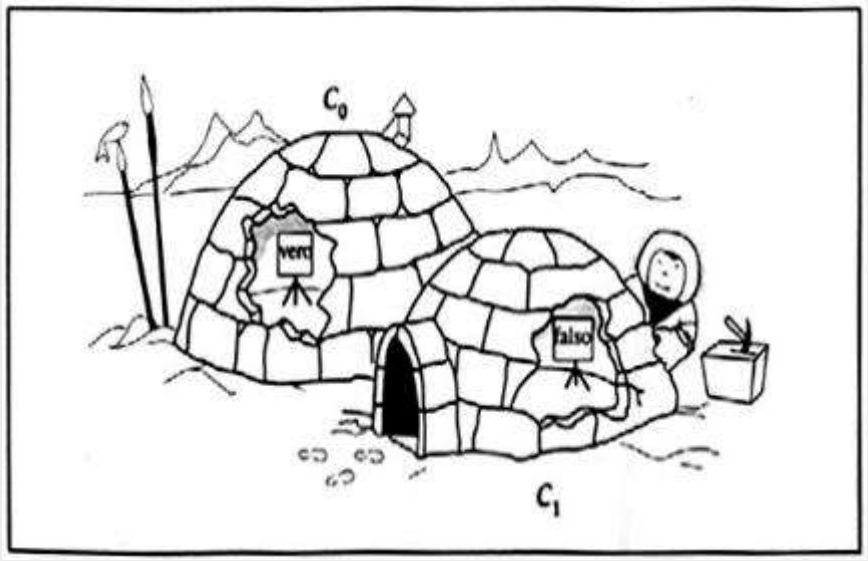

```
boolean flag[2];

//Processo 0
…
while (flag[1])
    {nulla;}
flag[0]=true;
<sezione critica>
flag[0]= false;
…
```

PRO: se un processo fallisce fuori della sua sezione critica l'altro può continuare a lavorare
CONTRO:

1. se un processo fallisce entro la sezione critica o prima di mettere false nel suo flag allora l'altro è bloccato per sempre.
2. se entrambi vedendo il flag dell'altro false, mettono il loro flag a true, entrambi vanno nella sezione critica, senza mutua esclusione. Dipendenza della velocità dei processi.

Algoritmo di Dekker
3° tentativo




```
boolean flag[2];


// Processo 0
…
flag[0] = true; //indico prima del
    //test di voler andare in sezione
    //critica
while (flag[1])
{nulla}
<sezione critica>;
flag[0] =  false;
…
```

PRO: la mutua esclusione è garantita

PROBLEMI:

- Se un processo fallisce entro la sua sezione critica l'altro è bloccato
- Se entrambi settano il flag a true prima che uno dei due verifichi la condizione del while si ha stallo

Algoritmo di Dekker
4° tentativo

Processo 0

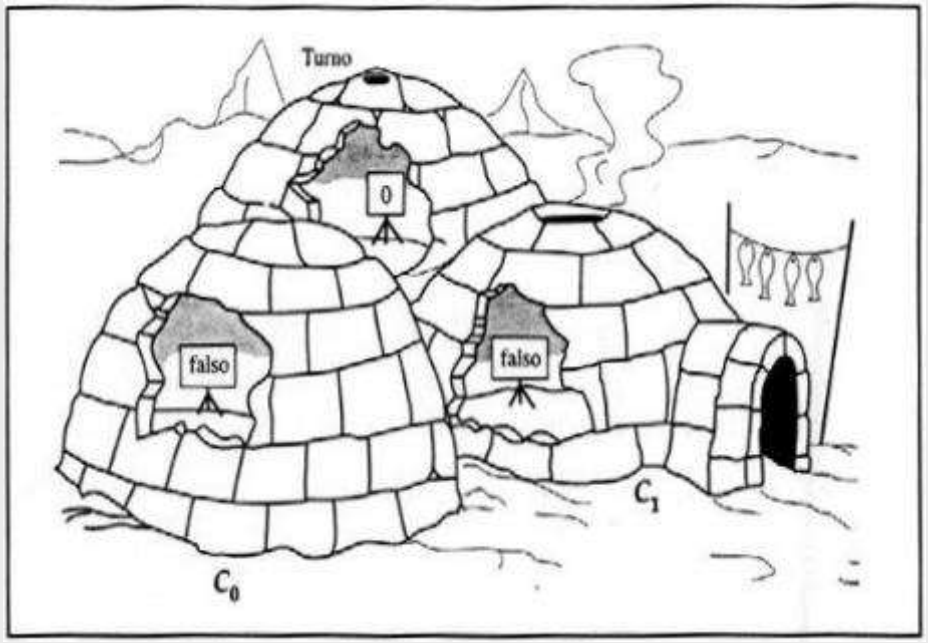
```
…
flag[0]= true;
while (flag[1])
{
   flag[0] = false;
   <pausa>
   flag[0] = true;
}
<sezione critica>;
flag[0] = false;
…
```

Processo 1

```
…
flag[1] = true;
while (flag[0])
{
   flag[1] = false;
   <pausa>;
   flag[1] = true;
}
<sezione critica>;
flag[1] = false;
…
```

PRO: mutua esclusione garantita
PROBLEMI:
- P0: "vado io", P1: "vado io", P0:"non vado io", P1: "non vado io", attesa….
in realtà non'è uno stallo poiché chi esce prima dal ciclo di attesa riesce ad entrare nella sezione critica
- Se un processo fallisce entro la sua sezione critica l'altro è bloccato

Algoritmo di Dekker
Una soluzione corretta



*Ogni processo ha una propria variabile **flag** che indica se vuole andare nella sezione critica o meno*

*Variabile **turno** specifica chi ha il diritto di insistere nel tentativo di entrare nella propria sezione critica*

Es.: *P0 vuole entrare (pone il suo flag a true)*
P0 controlla il flag di P1
se falso entra nella sezione critica
se vero controlla il turno
se turno è 0 (il suo) continua a controllare periodicamente il flag di P1
se turno è 1 pone il suo flag a falso lasciando il passo a P1

Algoritmo di Dekker
Una soluzione corretta

```
boolean flag[2];
int turno;
void main()
{       flag[0]=false;          flag[1]=false;          turno=1; //turno=0;
        …
        processo P0;
        processo P1;
        …
}

//processo P0
{
        …
                flag[0]=true;
                while(flag[1])
                {
                        if (turno==1)
                        {
                                flag[0]=false;
                                while(turno==1)
                                        {<nulla…ATTESA ATTIVA>}
                                flag[0]=true;
                        }
                 }
                <sezione critica>
                flag[0]=false;
        …
}
```

Algoritmo di Dekker
Una soluzione corretta

- Algoritmo “complesso”
- Attesa attiva: un processo controlla continuamente il flag dell’altro processo se pari a true
- Se un processo fallisce nella propria sezione critica l’altro processo rimane bloccato per sempre

Mutua Esclusione: Supporto Hardware

Macchine monoprocessore: i processi si alternano in esecuzione (EXE)
la mutua esclusione si può ottenere evitando l'interruzione di un processo attivando e disattivando gli interrupt (supporto hardware)

<disattiva le interruzioni>
<sezione critica>
<attiva le interruzioni>

PRO: mutua esclusione garantita
PROBLEMI:

1. Efficienza peggiora poiché il processore non può alternare i processi liberamente
2. Non funziona su macchine SMP

Soluzione su SMP

Istruzioni macchina speciali per l'accesso a locazioni di memoria in modo atomico (non interrompibile)

test-and-set
scambio (swap)

l'accesso sequenziale ad una locazione di memoria è garantita dall'hardware
Se si eseguono due test-and-set (swap) contemporaneamente, esse vengono serializzate

Mutua Esclusione: Supporto Hardware - test&set

```
boolean TestAndSet (boolean *target)
{
   boolean val = *target;
   *target = TRUE;
   return val;
}
```

- Utilizzo di test&set per garantire la mutua esclusione
  sia lock una variabile boolean condivisa inizializzata a falso (la risorsa è libera).

```
while (true) {
              while ( TestAndSet (&lock ))
              ;   /* do nothing
              <critical section>
              lock = FALSE; //rilascio
              …
              }
```

Mutua Esclusione: Supporto Hardware - swap

```
void swap (boolean *a, boolean *b)
          {
                  boolean temp = *a;
                  *a = *b;
                  *b = temp:
          }
```

- Utilizzo di swap per garantire la mutua esclusione
  sia lock una variabile booleana condivisa inizializzata a FALSE //indica risorsa accessibile

```
while (true)  {
                    key = TRUE;
                    while (key == TRUE)
                              swap (&lock, &key ); //in key torna false
                    <critical section>
                    lock = FALSE;
                    …
                }
```

Mutua Esclusione: Supporto Hardware

Vantaggi

- si può applicare a un qualsiasi numero di processi anche su multiprocessori a memoria condivisa;
- si può usare per gestire più di una sezione critica, ciascuna con una propria variabile;

Svantaggi

- Attesa attiva (i processi consumano tempo di cpu)
- Starvation (la scelta di quale processo andrà nella sezione critica è arbitraria)
- Stallo
  - Es (singolo processore)
    - P1 in sezione critica
    - P2 ha priorità più alta di P1
    - P2 tenterà di accedere (tramite test&set o swap) alla risorsa bloccata da P1 le nella sua attesa attiva non lascerà mai il posto a P1 che ha priorità più bassa

Mutua Esclusione: Supporto del SO e dei ling. di prog.

SEMAFORI

SEMAFORO: variabile (intera) sulla quale sono possibili 3 operazioni:

1. Inizializzazione ad un valore non negativo
2. Operazione atomica **wait()**: decrementa il valore della variabile. Se il valore della variabile diventa negativa, il processo che ha eseguito la wait viene bloccato.
3. Operazione atomica **signal()**: incrementa il valore della variabile. Se il valore della variabile è negativo, uno dei processi bloccati sull'operazione di wait viene sbloccato

- *Si associa un semaforo ad ogni risorsa condivisa*
- *Il processo che vuole utilizzare la risorsa effettua una operazione di wait*
- *Il processo che rilascia la risorsa effettua il signal*
- *La variabile numerica indica il numero di istanze di una specifica risorsa condivisa (semaforo contatore)*
- *Se la variabile è negativa, essa rappresenta (presa in valore assoluto) il numero di processi in attesa*

```
wait (S);
<Critical
Section>
signal (S);
```

Implementazione dei semafori Contatore

```
typedef struct {
                int istanze;
                struct processo *P; //lista dei processi in coda
                } semaforo;
```

```
void wait(semaforo s)
{
        s.istanze--;
        if(s.istanze<0)
                {
                        <poni processo in coda>
                        <blocca questo processo: running->blocked >
                }
}
```

```
void signal(semaforo s) :
{
        s.istanze++;
        if(s.istanze<=0)
                {
                        <rimuovi un processo in coda>
                        <sveglia il processo: blocked->ready >
                }
}
```

Implementazione dei semafori binari

Semaforo binario: semaforo il cui valore intero può essere solo 0 o 1
Gestione più complessa che non con semafori contatore

```
typedef struct {
                    boolean val;
                    struct processo *P; //lista dei processi in coda
                    } semaforo_bin;
```

```
void wait(semaforo_bin s)
{
          if (s.val==1)
                    s.val=0;
          else
                    {         <poni processo in coda a P>
                              <blocca questo processo: running->blocked >
                    }
}
```

```
void signal (semaforo_bin s)
{
          if(*P==NULL) //coda vuota
                    s.val=1;
          else
                    {         <rimuovi un processo in coda>
                              <sveglia il processo: blocked->ready >
                    }
}
```

Implementazione dei semafori …

Come fare affinchè signal e wait siano atomiche??

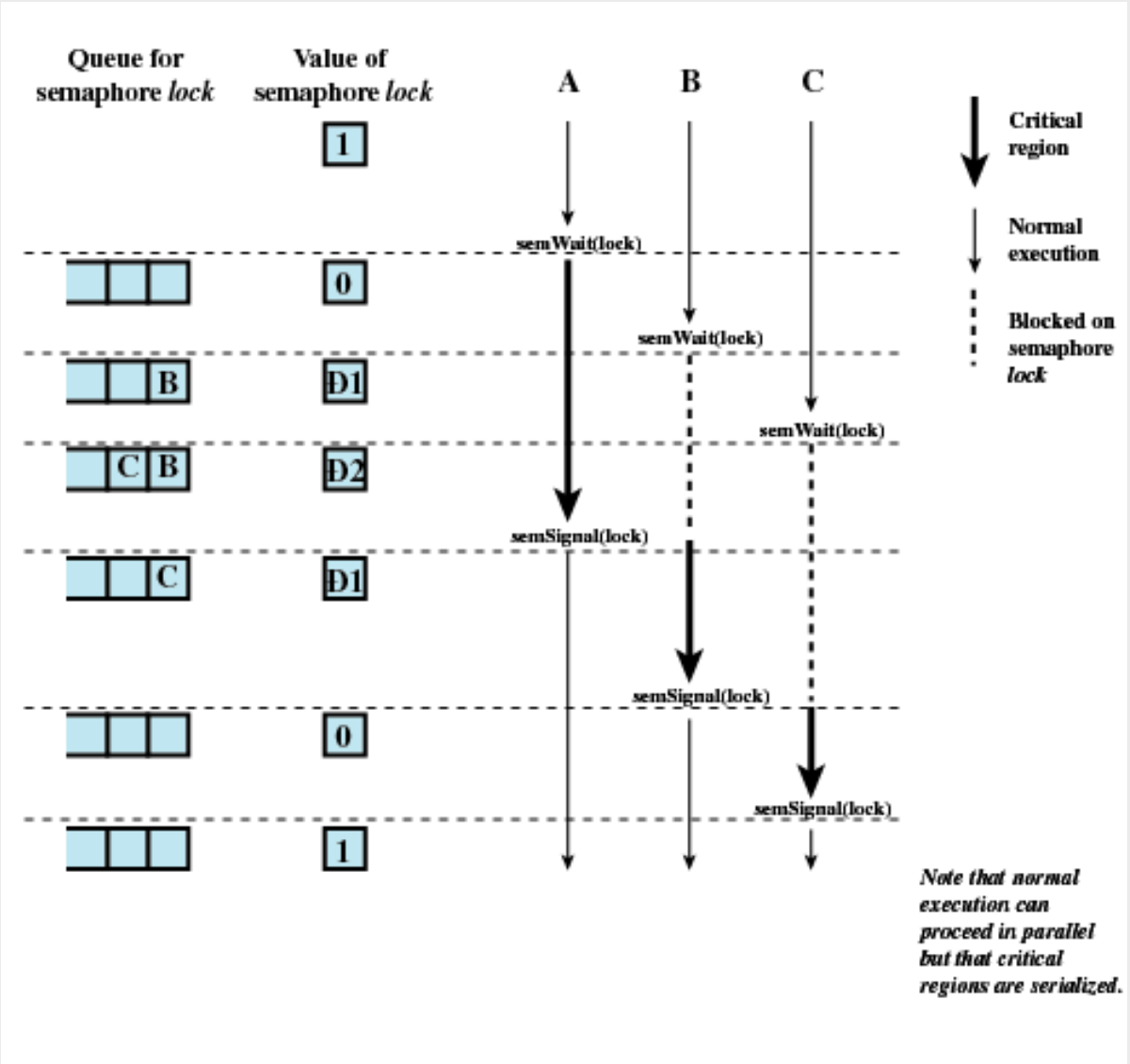
1. **Implementarle in hardware o firmware**

2. Dekker o Peterson…sovraccarico di elaborazione

3. **Utilizzo dell'istruzione atomica test&set**
   Esercizio: si scriva lo pseudocodice C

4. Se il sistema è monoprocessore basta disabilitare gli interrupt durante le operazioni
   Esercizio: si scriva lo pseudocodice C

Accesso a dato condiviso tramite l'uso dei semafori



Deadlock e Starvation

- Deadlock – due o più processi sono in attesa di un evento che può essere determinato solo da uno dei processi in attesa

Siano S e Q due semafori inizializzati a 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- *P1 non rilascia S fino a quandonon ottiene Q*
- *P0 non rilascia Q fino a quando non ottiene S*

Le signal non saranno mai eseguite: STALLO

- Starvation – indefinite blocking. Un processo non viene mai rimosso dalla coda al semaforo. Si immagini una gestione LIFO.

Produttori e Consumatori
- Situazione tipica di processi concorrenti -

Tipica rappresentazione dei processi concorrenti:

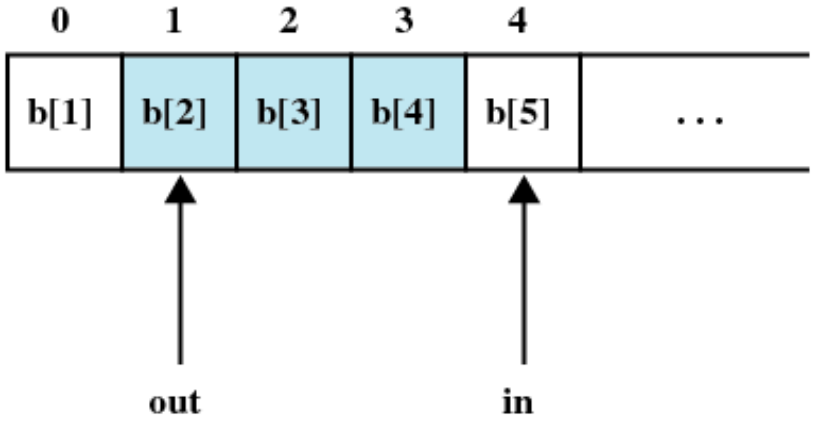
Uno o più produttori generano dati inserendoli in un buffer

Un consumatore preleva i dati uno alla volta

- Ipotesi di buffer infinito: l'accesso al buffer tra produttore e consumatore deve essere mutuamente esclusivo

```
produttore
        <produce dato>
        buffer[in]=dato;
        in++;
```

```
consumatore
   while(in >= out)
   { w = buffer[out];
     out++;
     <consuma w>
   }
```



Produttori e Consumatori
Buffer infinito – semafori binari

```
/* program producerconsumer */
int n;  //Numero di elementi nel buffer
binary_semaphore s = 1; //gestisce l'accesso al buffer
binary_semaphore delay = 0; //gestisce il caso di nessun
                            //elemento nel buffer
void producer()
{
     while (true)
     {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)
     {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

…con semafori binari…

Essendo il buffer infinito, il produttore può inserire tutto quello che produce

Se n=1, significa che il buffer prima era vuoto ed occorre avvisare il consumatore

Si pone in attesa che il primo elemento venga prodotto

Se il consumatore ha svuotato il buffer, si mette in attesa che venga prodotto un nuovo elemento

Produttori e Consumatori
Buffer infinito – semafori contatore

```
/* program  producerconsumer */
semaphore n = 0;//Gestisce il caso di nessun elemento nel  buffer
semaphore s = 1;  //Gestisce l'accesso al buffer
void producer()
{
     while (true)
     {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true)
     {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.11 A Solution to the Infinite-Buffer Producer/Consumer Problem

…con semafori contatore…

È indifferente il loro ordine???
SI

È indifferente il loro ordine???
NO..
Il consumatore entrerebbe nella sezione critica quando il buffer è vuoto e nessun produttore potrebbe aggiungere elementi: STALLO

MONITOR

I semafori sono primitive potenti per gestire la mutua esclusione, ma la scrittura di un programma con l'utilizzo dei semafori può essere tutt'altro che semplice.

MONITOR:

- costrutto di sincronizzazione di alto livello
- modulo software le cui caratteristiche principali sono:
  - Variabili locali accessibili solo dalle procedure (metodi) definite al suo interno e non da procedure esterne
  - Un processo entra nel monitor chiamando le sue procedure
  - Un solo processo alla volta può essere in esecuzione all'interno del monitor. Gli altri processi sono sospesi nell'attesa che il monitor diventi disponibile

- UTILIZZO:
  - Mutua esclusione
  - Protezione delle strutture dati condivise

- Sincronizzazione tramite variabili di condizione. Su tali variabili si opera mediante:
  - *cwait (c)* : sospende l'esecuzione del processo chiamante sulla condizione c
  - *csignal (c)* : riattiva un processo sospeso sulla condizione c. NB: se non c'è nessun processo in attesa su tale condizione il segnale viene perso