Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2017-18

Pietro Frasca

Lezione 11

Giovedì 9-11-2017

- Molti processori possiedono istruzioni che consentono di leggere e modificare il contenuto di una locazione in un unico ciclo di memoria. Un esempio è dato dall'istruzione **TSL** (**Test and Set Lock**).
- L'istruzione **TSL R, X** copia il contenuto della locazione di memoria **X** nel registro **R** del processore e viene scritto in X un **valore diverso da 0**.
- Nel caso di sistemi multiprocessore, il processore che esegue la TSL **blocca il bus di memoria** per impedire che altri processori accedano alla memoria fino a quando non ha completato l'operazione di TSL.
- La mutua esclusione si ottiene realizzando due funzioni **lock(x)** e **unlock(x):**

- **Lock(x):**

```
LOCK:
   TSL R, X    copia il valore di X in R e pone X=1
               (R=X;X=1)
   CMP R,0     verifica se R==0
   JNE LOCK    se R!=0 riesegue il ciclo
   RET         ritorno
```

- **Unlock(x)**

```
  mov x,0      scrive in X il valore 0
  RET          ritorna al chiamante
```

Esempio di due processi

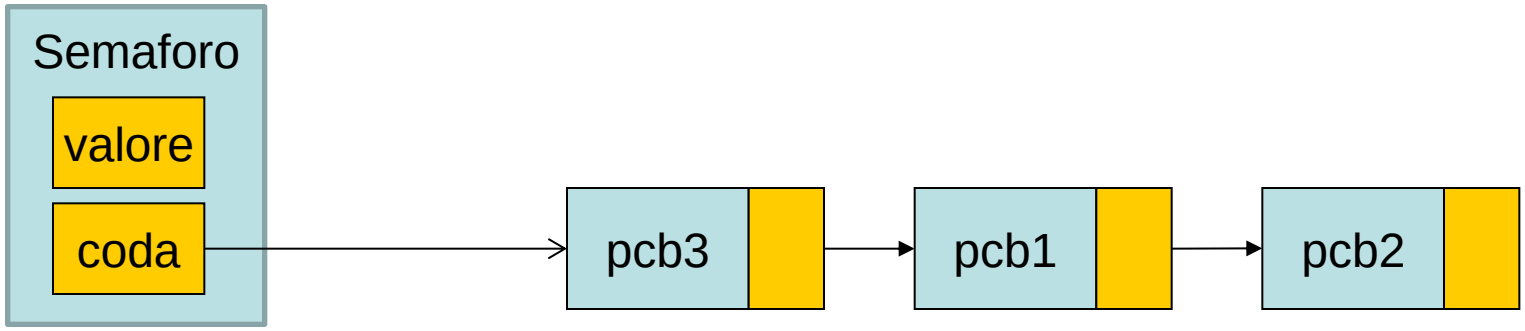
```
P1
Prologo:   lock(x)
           <sezione critica P1>
Epilogo:   unlock(x)

P2
Prologo:   lock(x)
           <sezione critica P2>
Epilogo:   unlock(x)
```

Questa soluzione è caratterizzata da condizioni di **attesa attiva** dei processi. La soluzione è valida quindi per sistemi multiprocessore ed è limitata al caso di sezioni critiche **brevi**.

Semafori

- un semaforo **s** è una struttura dati gestita principalmente dalle funzioni **wait(s)** e **signal(s) e** dalla funzione di inizializzazione **init(s,valore).**
- La struttura dati è costituita da una variabile intera non negativa ***valore*** e da una coda di descrittori di processi sospesi ***coda.***

```
typedef struct {
    int valore;
    struct processo *coda;
} semaforo;
```



- *La **wait()*** è chiamata da un processo per verificare lo stato di un semaforo secondo il seguente pseudocodice:

```
void wait(semaforo s){
  if (s.valore==0) {
    <il processo viene sospeso>
    <il descrittore del processo viene inserito in
     s.coda>
   }
   else
     s.valore=s.valore-1;
}
```

- Se **s.valore** = 0, la *wait()* porta il processo nello **stato di bloccato** e inserisce il suo descrittore nella coda **s.coda** associata al semaforo.
- Se **s.valore** è > 0, esso viene decrementato di 1 e il processo continua la sua esecuzione;
- La primitiva ***signal()*** risveglia eventuali processi che si trovano sospesi sul semaforo:

```
void signal(semaforo s) {
  if ( <se la coda s.coda non è vuota>)
     <estrai dalla prima posizione di s.coda il
      descrittore del processo portandolo nello
      stato di pronto>
   }
   else
     s.valore=s.valore+1;
}
```

- La ***signal()*** **non è bloccante** per il processo che la chiama, mentre la ***wait()*** è **bloccante** se **s.valore=0**.
- Le chiamate *wait()* e *signal()* devono essere realizzate in modo che siano eseguite in modo indivisibile. L'atomicità delle funzioni *wait()* e *signal()* si realizza a livello di kernel **disabilitando le interruzioni** del processore durante la loro esecuzione.
- Il semaforo è stato ideato da Dijkstra, e usato per la prima volta nel sistema operativo Theos.
- Il nome originale della *wait()* era ***P*** e quello della *signal()* era ***V***. Tali nomi erano stati attribuiti dallo stesso Dijkstra, e corrispondono alle iniziali delle parole olandesi *proberen* (verificare) e *verhogen* (incrementare).

Soluzione al problema della mutua esclusione con semafori.

- Si associa alla risorsa condivisa un semaforo **mutex** **inizializzandolo al valore 1** (**libero**) e usando per ogni processo che richiede la risorsa il seguente protocollo:

```
Init(mutex,1);
Prologo:  wait(mutex);
            <sezione critica>;
Epilogo:  signal(mutex);
```

- Esempio di due processi P1, P2 che accedono alla stessa risorsa comune R. Tale schema è valido per qualsiasi numero di processi.

```
P1
Init(mutex,1);
Prologo: wait(mutex);
         <sezione critica P1>;
Epilogo: signal(mutex);

P2
Prologo: wait(mutex);
         <sezione critica P2>;
Epilogo: signal(mutex);
```

- La soluzione mostrata evita condizioni di attesa attiva in quanto un **processo viene sospeso** se trova il semaforo occupato.
- Generalmente, la coda associata al semaforo è gestita con politica FCFS per evitare che qualche processo che si trova sospeso possa entrare in una situazione di attesa indefinita (starvation).
- Il semaforo che può assumere solo i due valori 0 e 1 prende il nome di **semaforo binario**, e spesso viene chiamato **mutex** (mutua esclusione).
- La **correttezza** della soluzione dipende dal **valore iniziale del semaforo** che deve essere posto a **1** e al corretto posizionamento delle funzioni di sistema *wait()* e *signal()* prima e dopo la sezione critica.

- Nei **sistemi multiprocessore**, per garantire che *wait()* e *signal()* siano eseguite in mutua esclusione sul semaforo, è necessario che i processi utilizzino le funzioni *lock()* e *unlock()*, secondo il protocollo seguente:

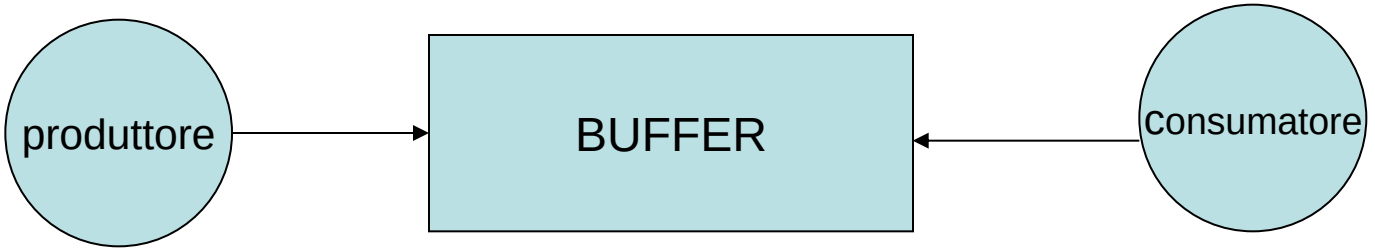
```
lock(x);
    wait(mutex);
unlock(x);
    <sezione critica>;
lock(x);
    signal(mutex);
unlock(x);
```

- La *lock()* garantisce che le chiamate *wait()* e *signal()* siano eseguite da un processo alla volta.

- La *wait()* e la *signal()***,** relative al semaforo ***mutex*****,** assicurano la mutua esclusione delle sezioni critiche su una **risorsa *R***, mentre la variabile *x*, con le *lock(x)* e *unlock(x)* assicura la mutua esclusione delle primitive *wait()* e *signal()* sul semaforo *mutex*.

Comunicazione e sincronizzazione tra processi

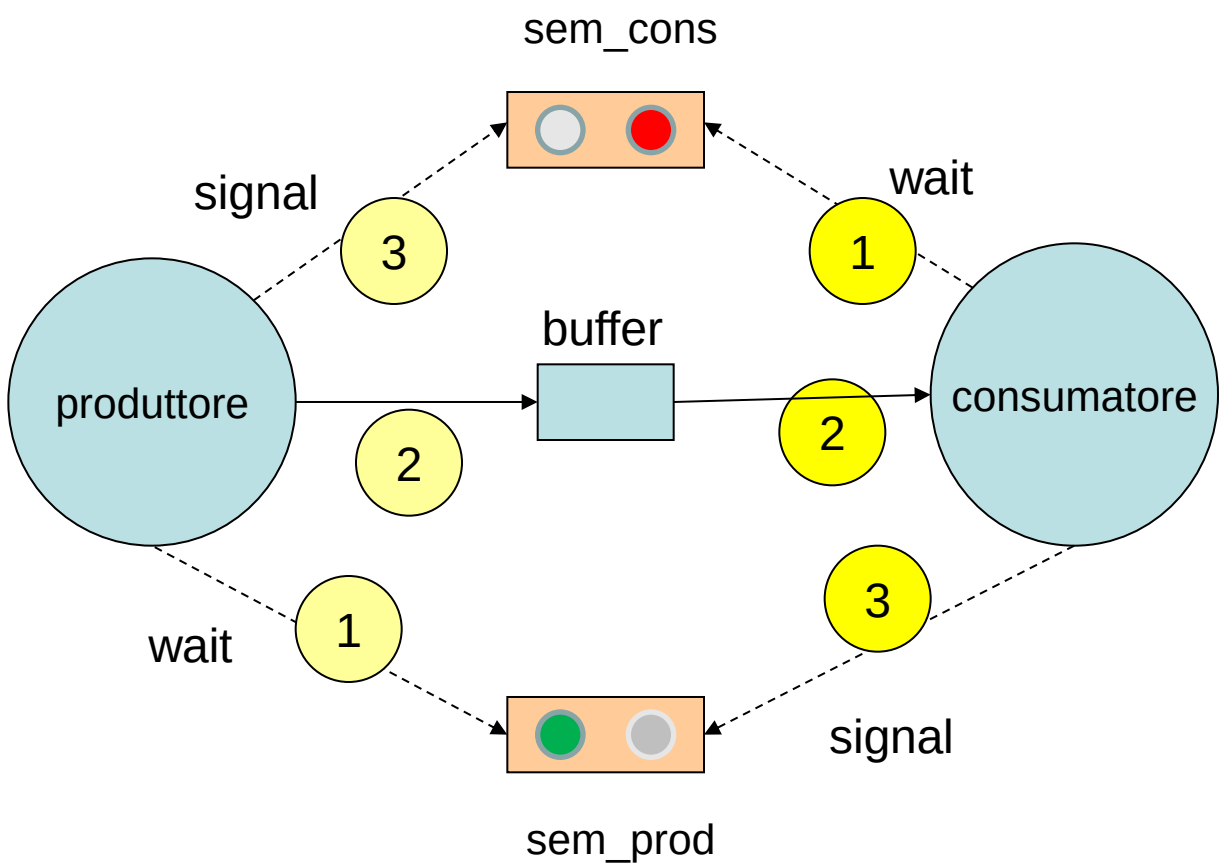
- Il paradigma del **produttore-consumatore** è spesso usato per la comunicazione tra processi.
- In tale modello, un processo detto ***produttore*** genera un messaggio e lo scrive in un area di memoria (buffer) che contiene un solo messaggio alla volta. Un processo, detto ***consumatore*** preleva dal buffer il messaggio e lo elabora.
- I processi devono accedere alla risorsa condivisa (il buffer) sia in mutua esclusione che eseguire le operazioni nel giusto ordine temporale. Per ottenere l'ordinamento è necessario che i due processi si scambino segnali: il produttore deve informare il consumatore che ha scritto un messaggio nel buffer, mentre il consumatore deve avvisare il produttore di aver letto il messaggio. Una soluzione a tale problema si ottiene ricorrendo ai **semafori**.



Soluzione al problema della comunicazione con semafori

Soluzione del problema del produttore-consumatore con buffer di capacità 1, utilizzando i semafori è la seguente:

- Si assume che il **buffer sia inizialmente vuoto**.
- Si utilizzano due semafori di nome ***sem_prod*** e ***sem_cons*** con le condizioni iniziali:
  - `Sem_prod.valore=1` (inizialmente il buffer è vuoto)
  - `Sem_cons.valore=0` (inizialmente non è presente alcun messaggio)



Sezione critica 2

```
buffer=x;
```

Sezione critica 2

```
x=buffer;
```

produttore-consumatore con buffer di capacità 1

```
void produttore (){
  do {
     <produzione nuovo messaggio>
     wait (sem_prod);
     <inserimento del messaggio nel buffer>
     signal(sem_cons);
  } while (!fine);
}

void consumatore () {
  do {
      wait (sem_cons);
      <prelievo del messaggio dal buffer>
      signal(sem_prod)
      <consumo del messaggio>
   } while (!fine);
}
```

Soluzione al problema della comunicazione con buffer di capacità N

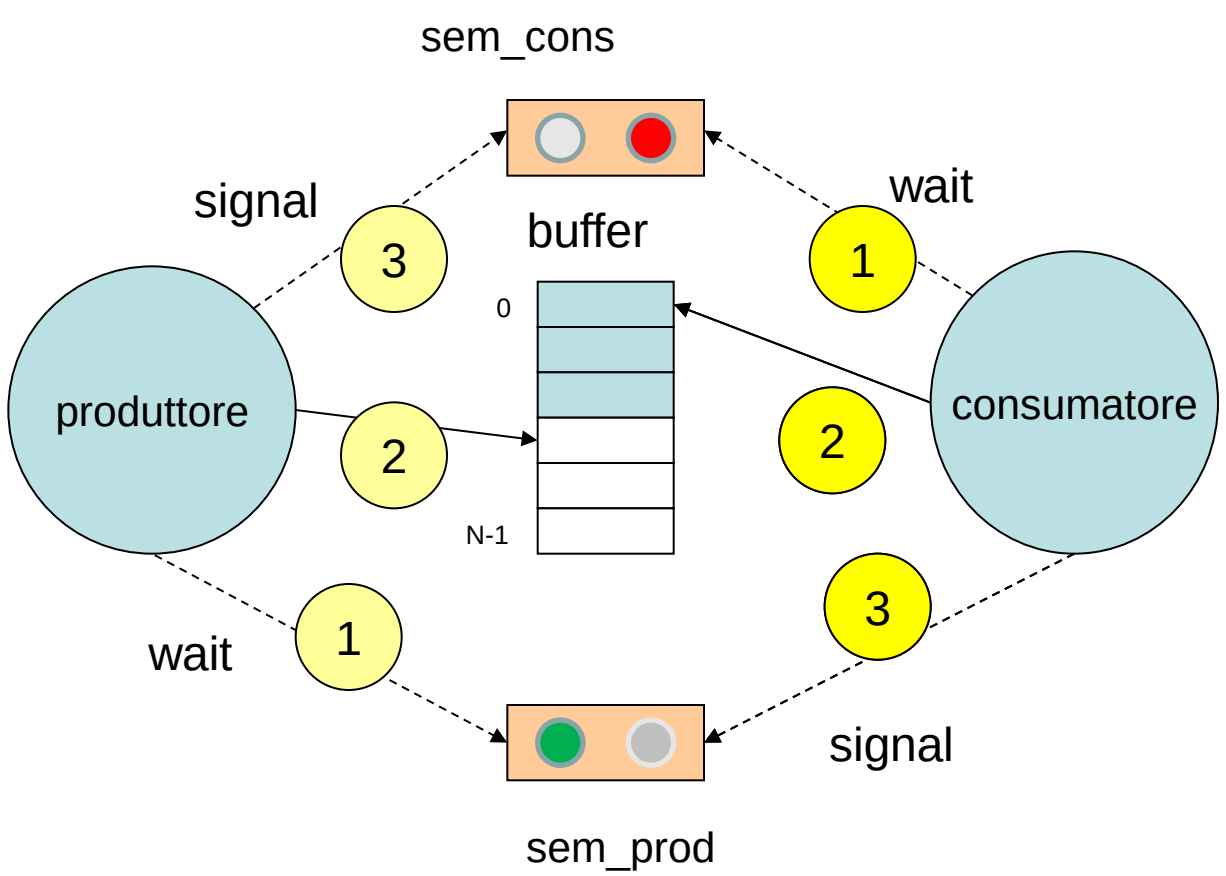
Soluzione del problema del produttore-consumatore con buffer di capacità N, utilizzando i semafori è la seguente:

- Il buffer è organizzato come un vettore circolare e gestito tramite due indici: **scrivi** che indica il prossimo elemento del buffer che sarà scritto dal produttore; **leggi** che indica il prossimo elemento che sarà letto dal consumatore.
  Inizialmente sarà:

  scrivi=leggi=0.
- Per sincronizzare l'accesso al buffer utilizziamo due semafori di nome **sem_prod** e **sem_cons** con le condizioni iniziali:

  sem_prod.valore=N;

  sem_cons.valore=0;.



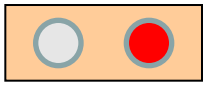
Sezione critica 2

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

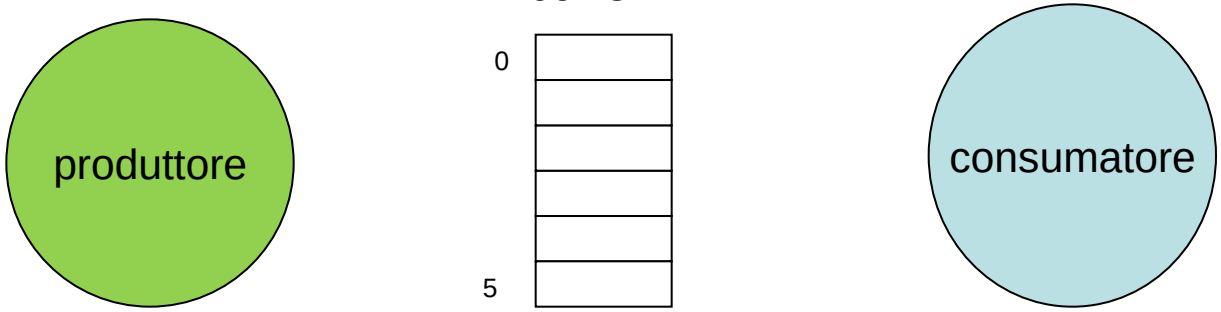
Sezione critica 2

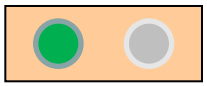
```
x=buffer[leggi];
leggi=(leggi+1)%N
```

sem_cons.valore=0



buffer





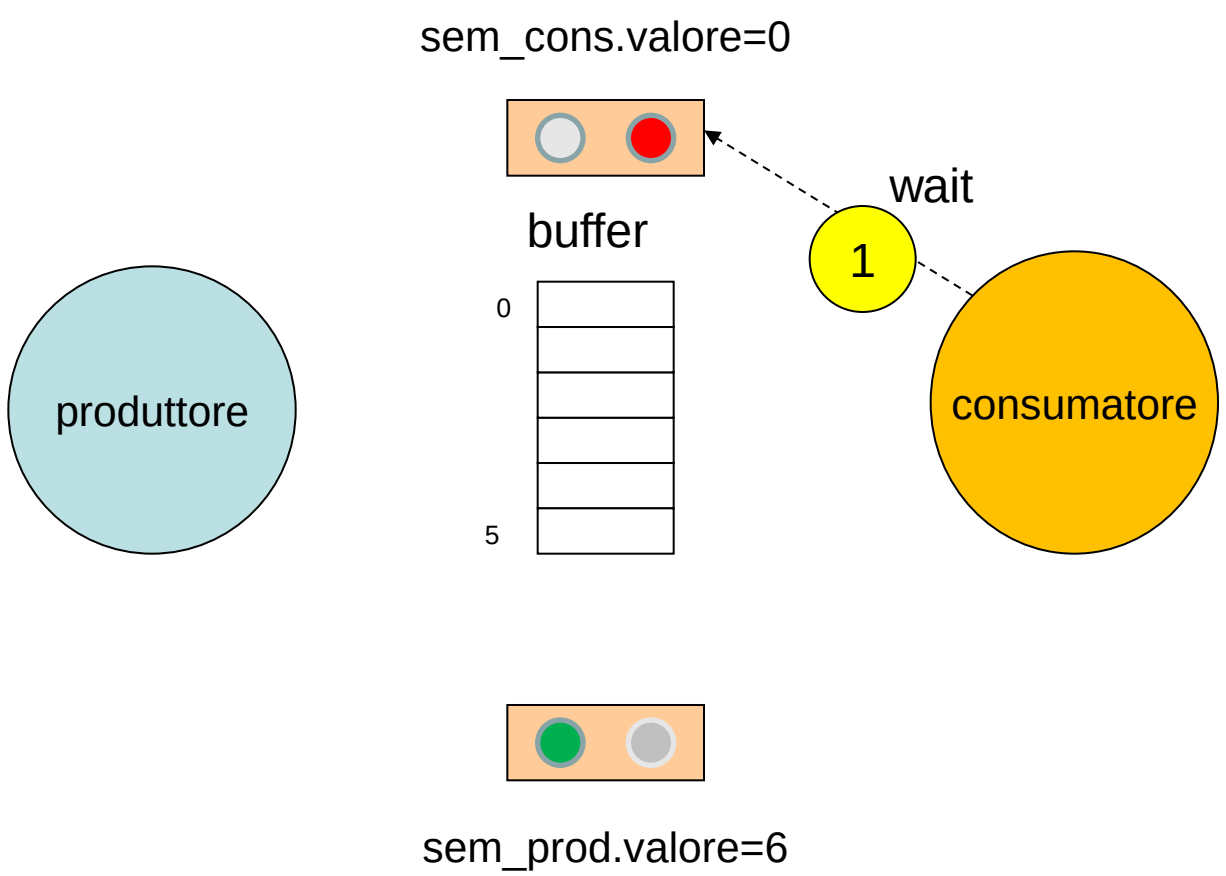
sem_prod.valore=6

Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

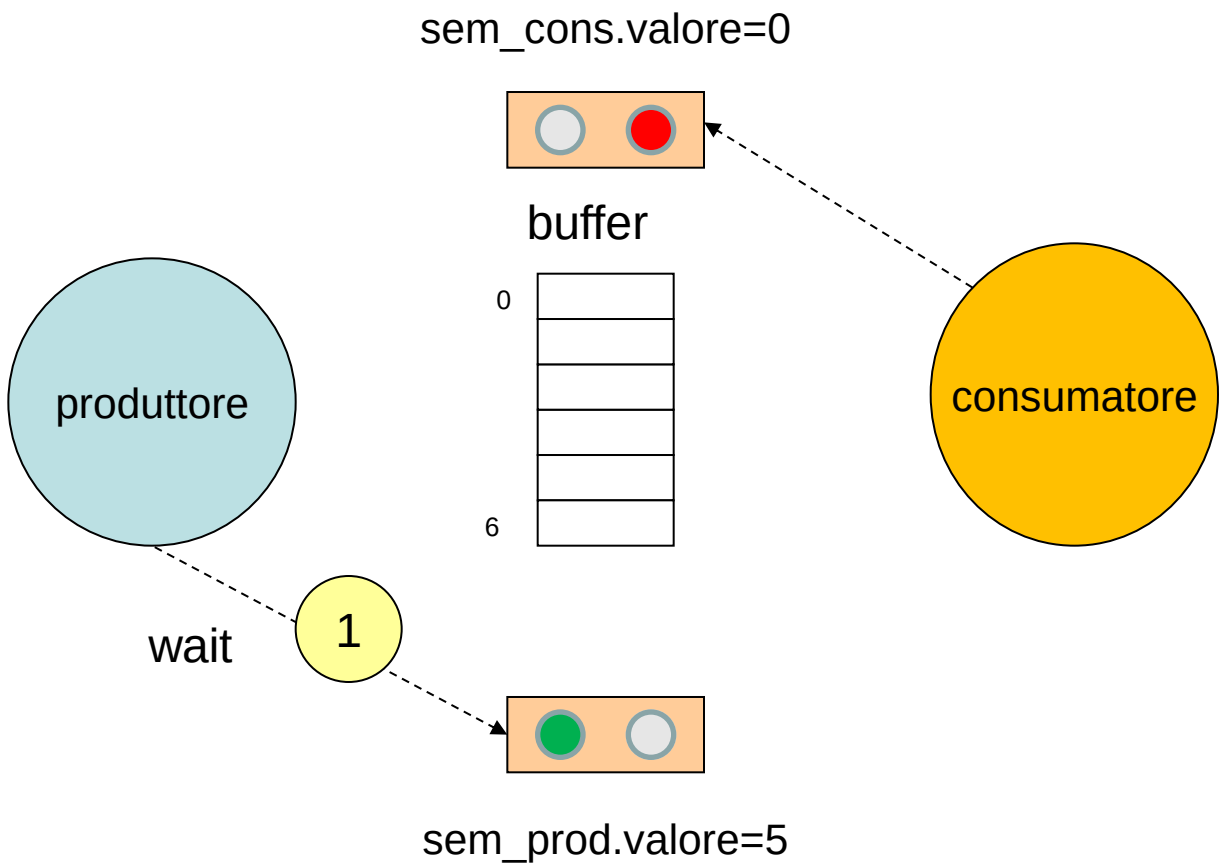


Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

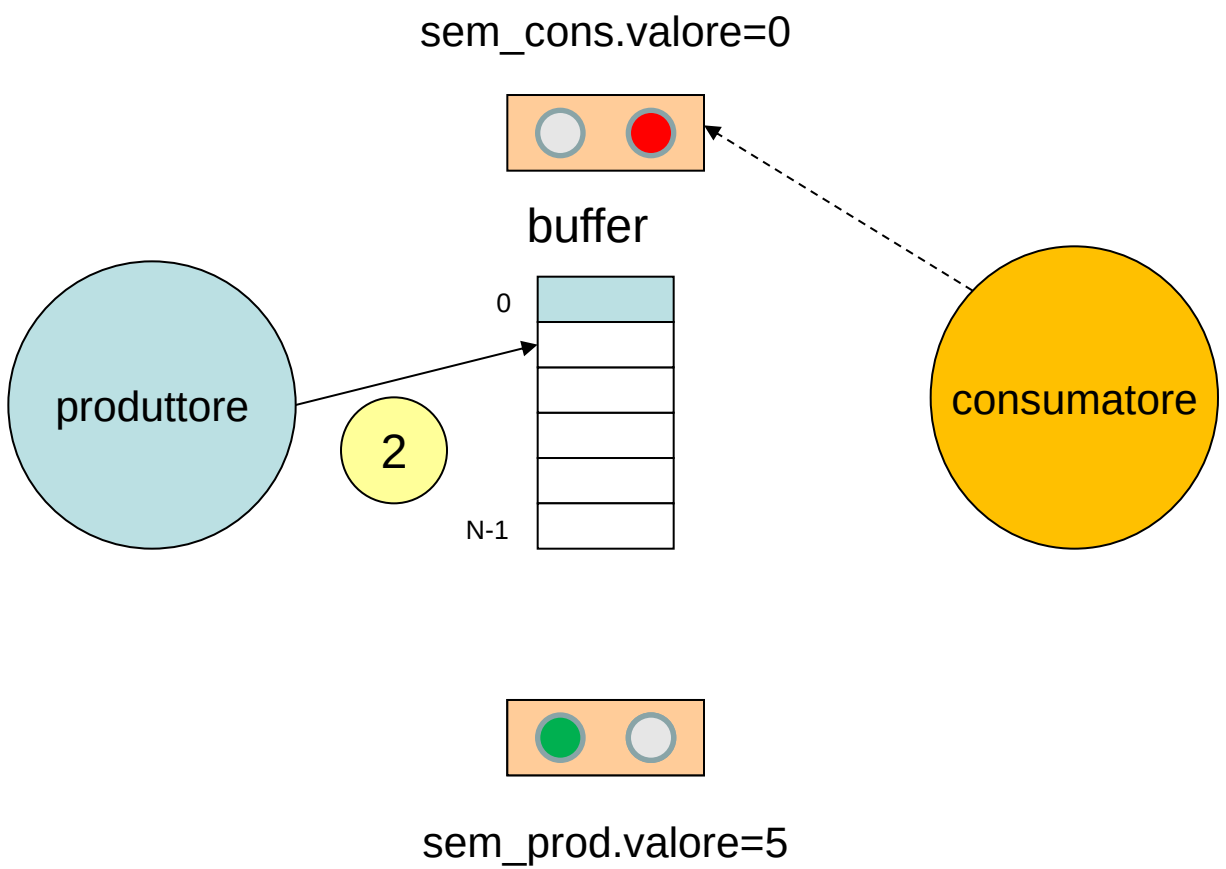


Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

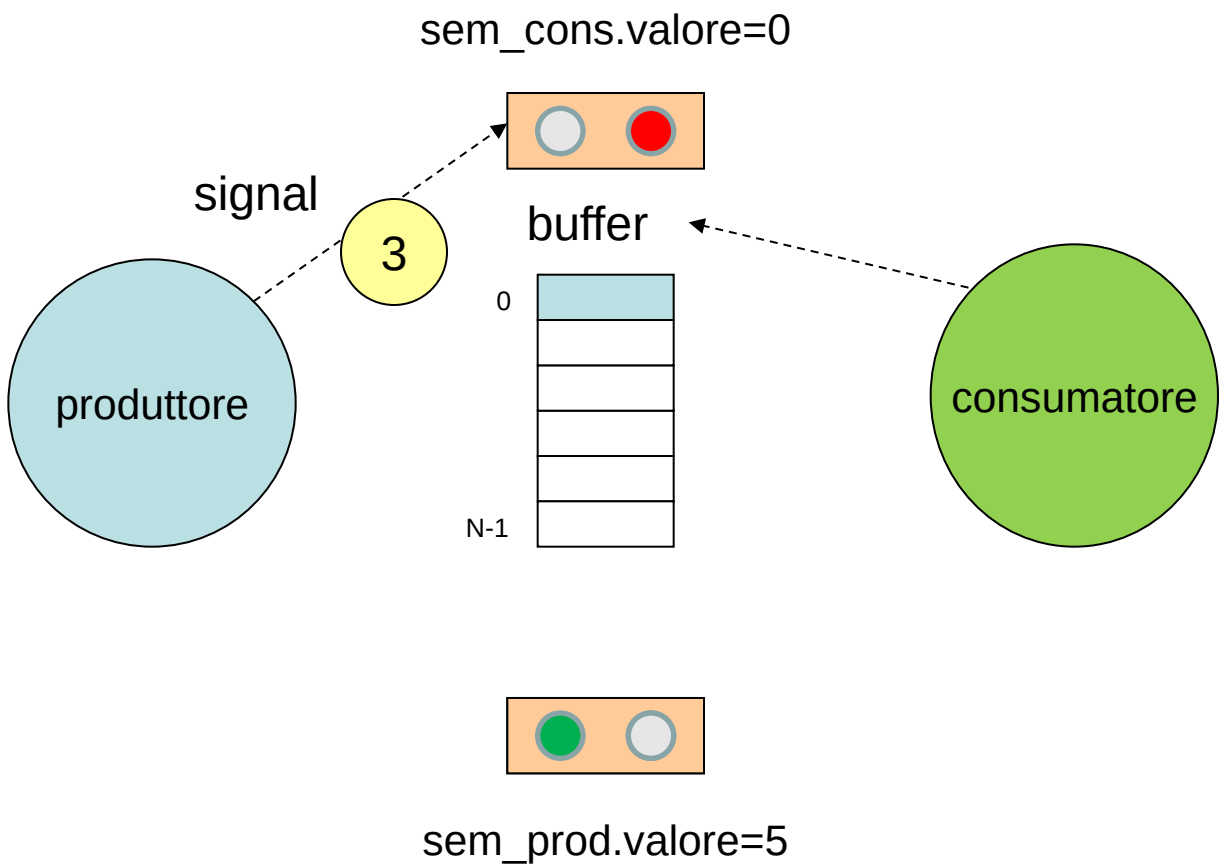


Sezione critica 2

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

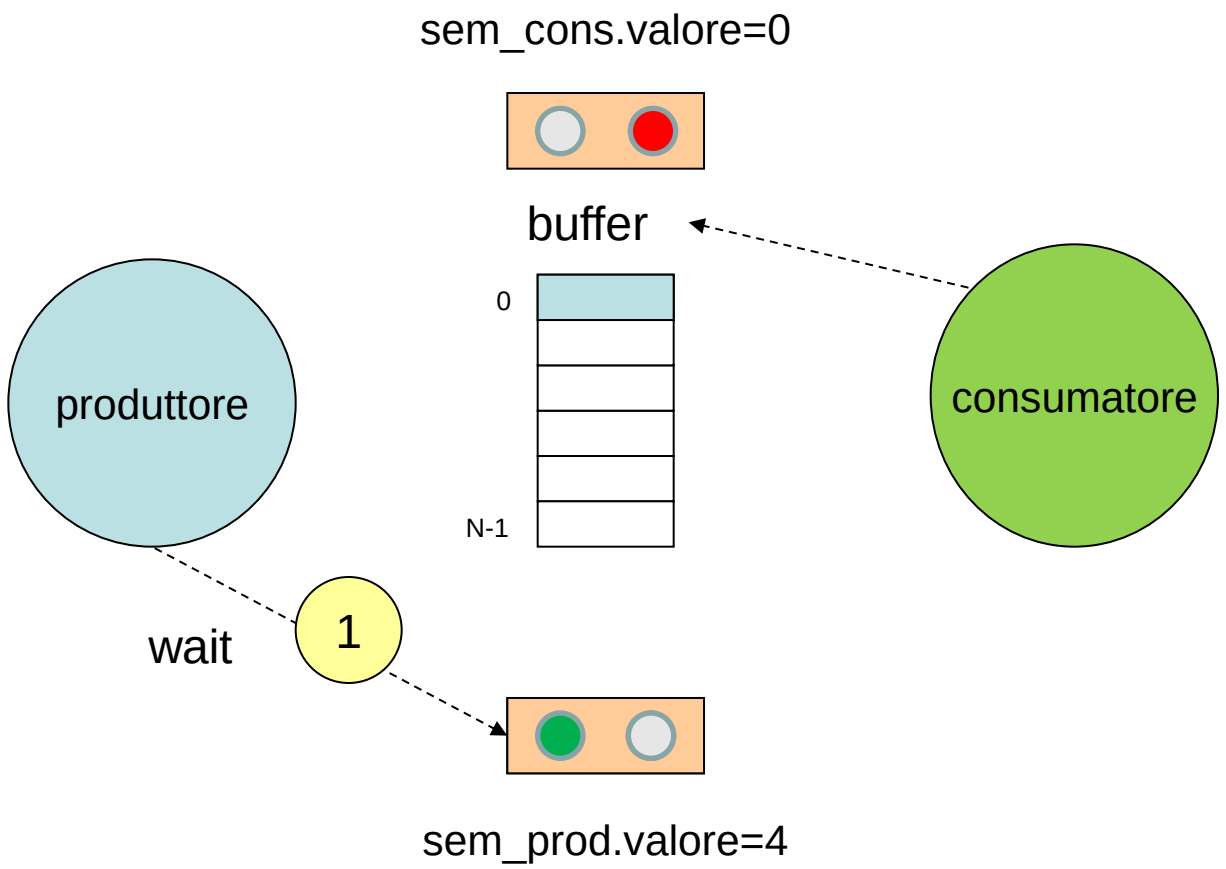


Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

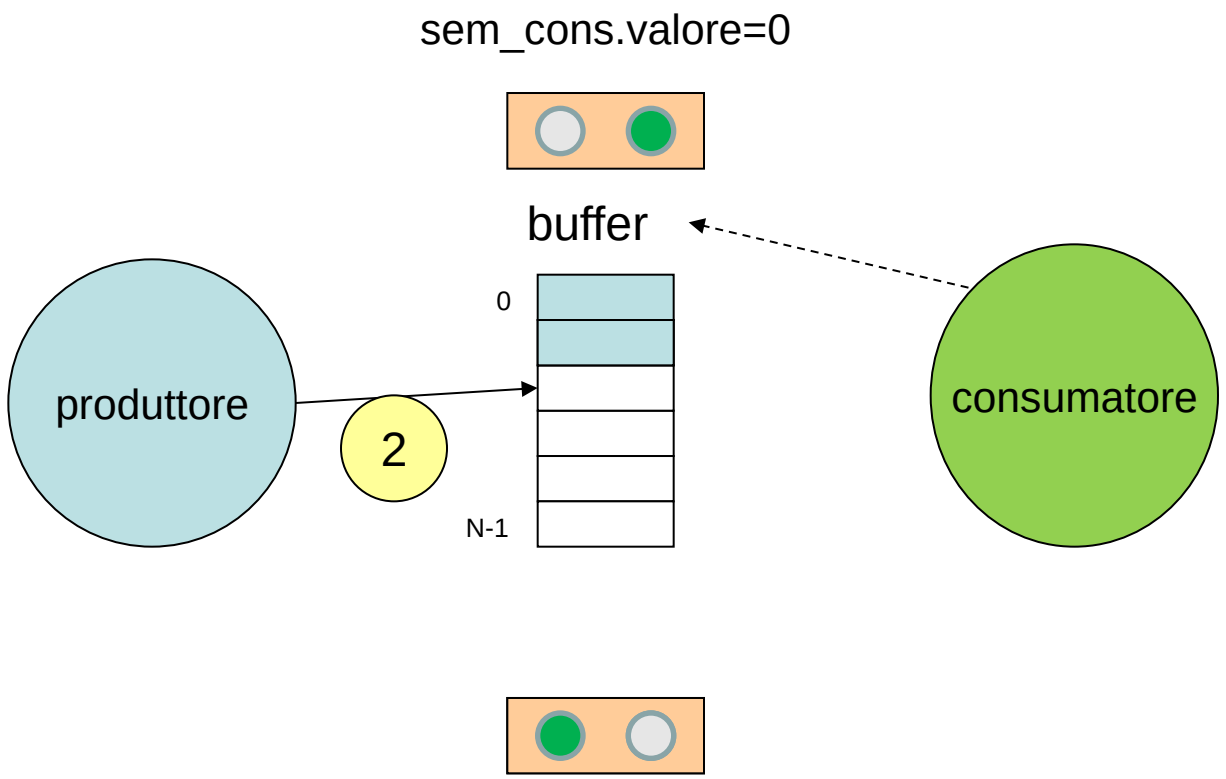


Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

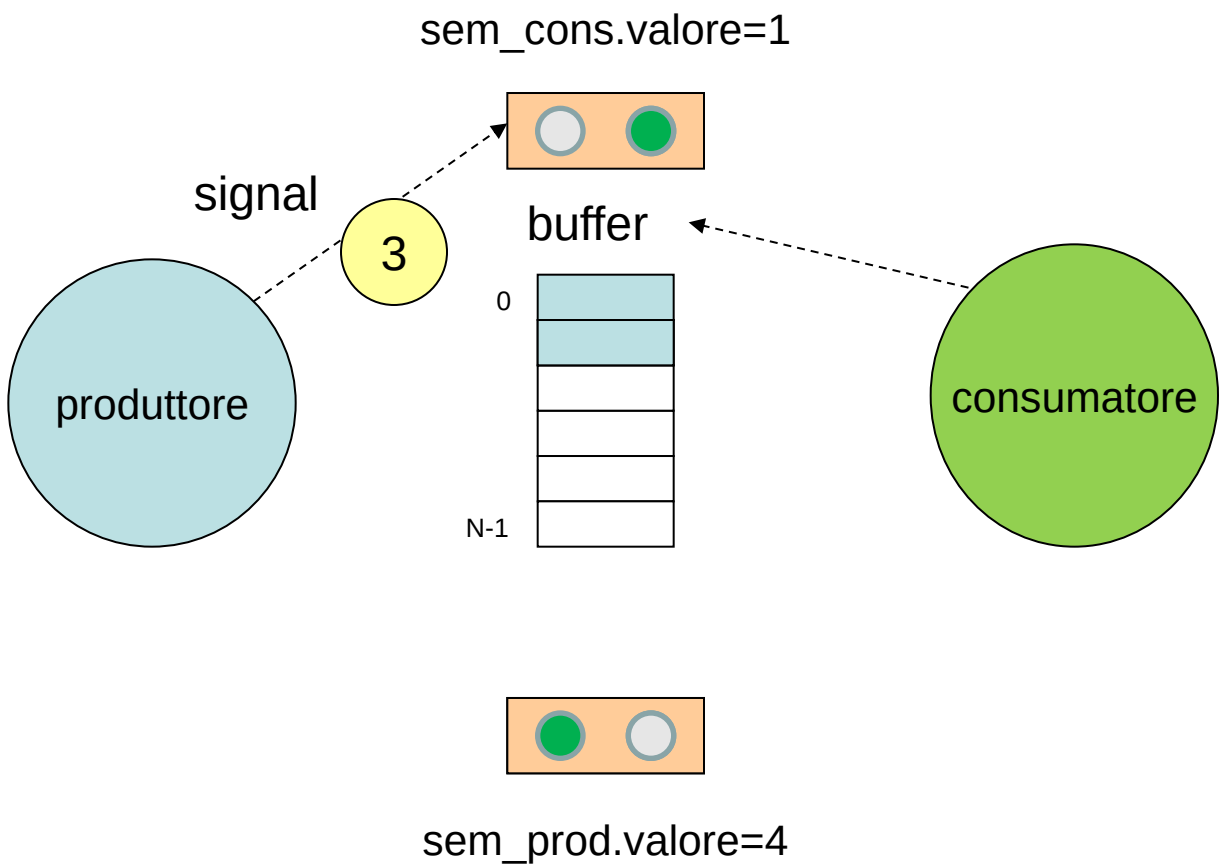


Sezione critica 2

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

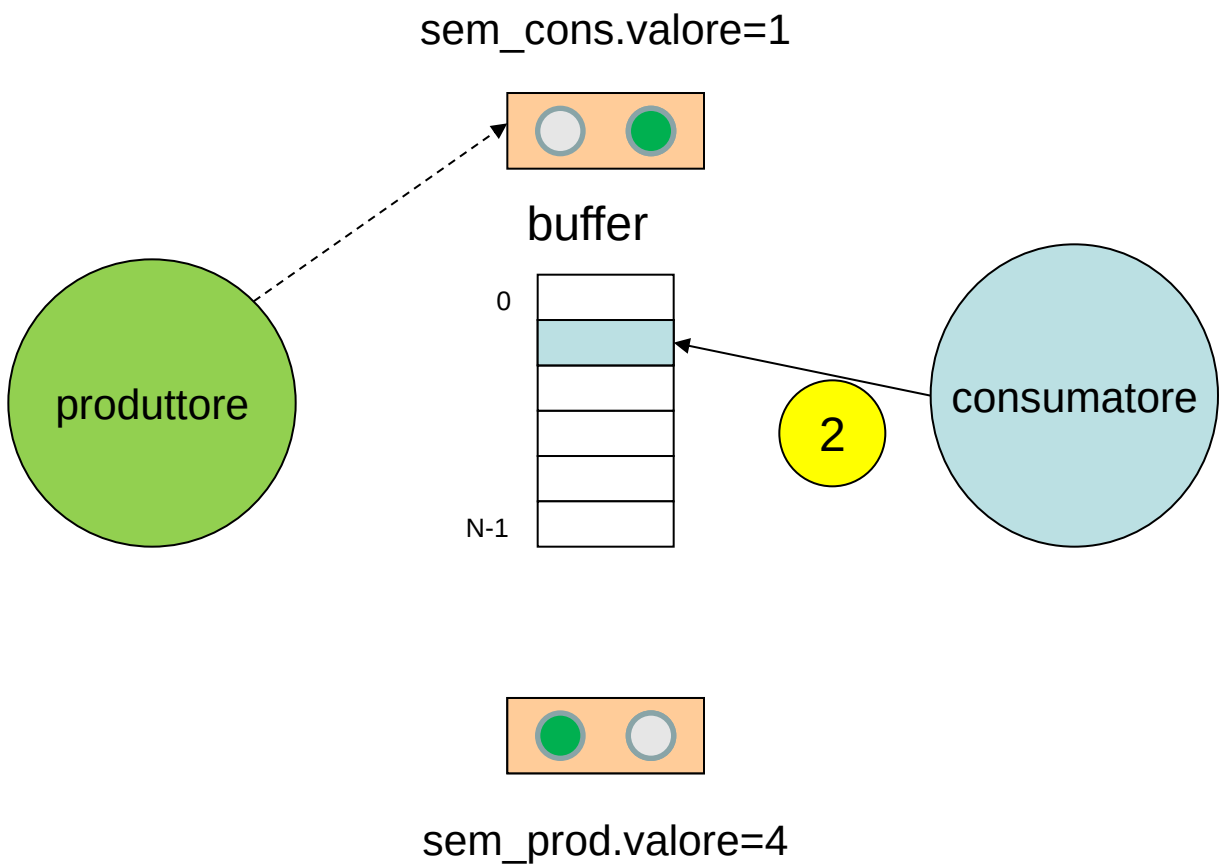


Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

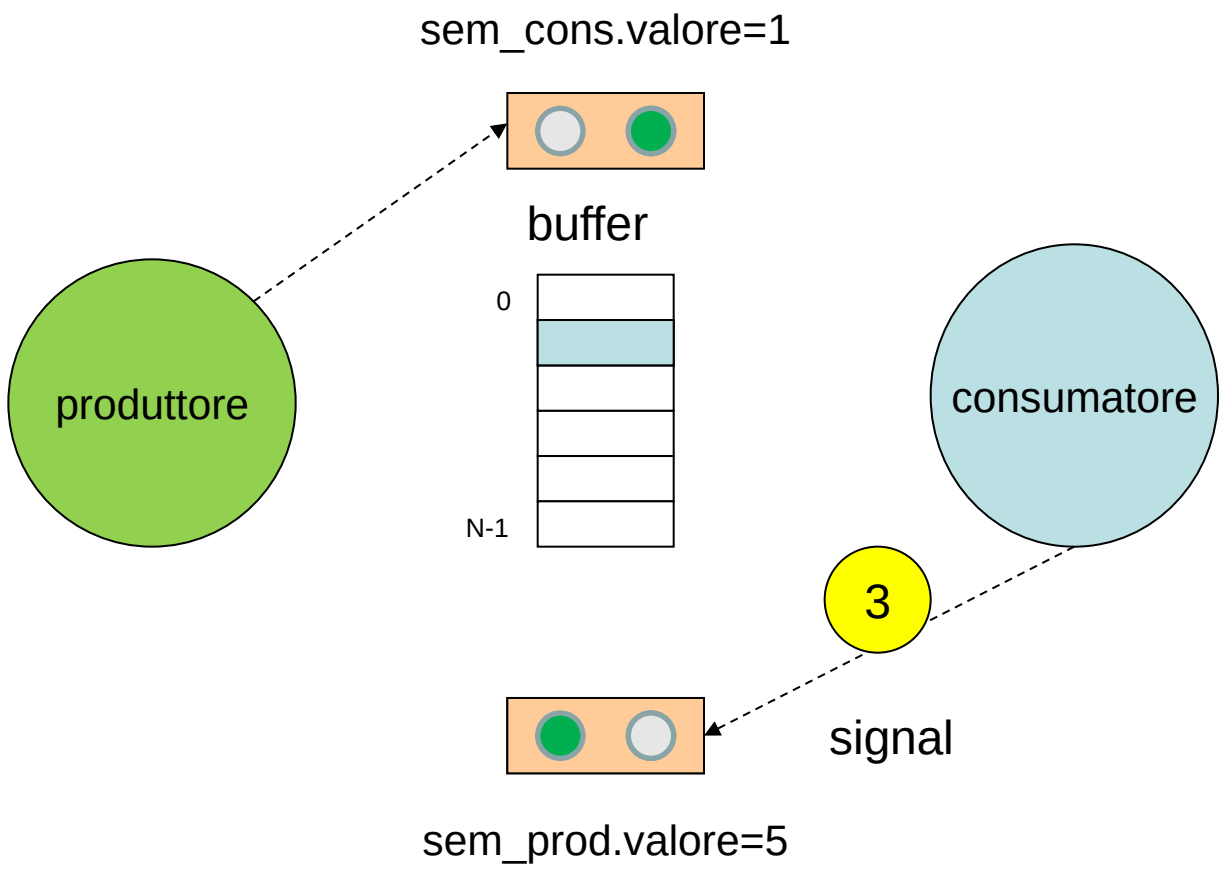


Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica 2

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

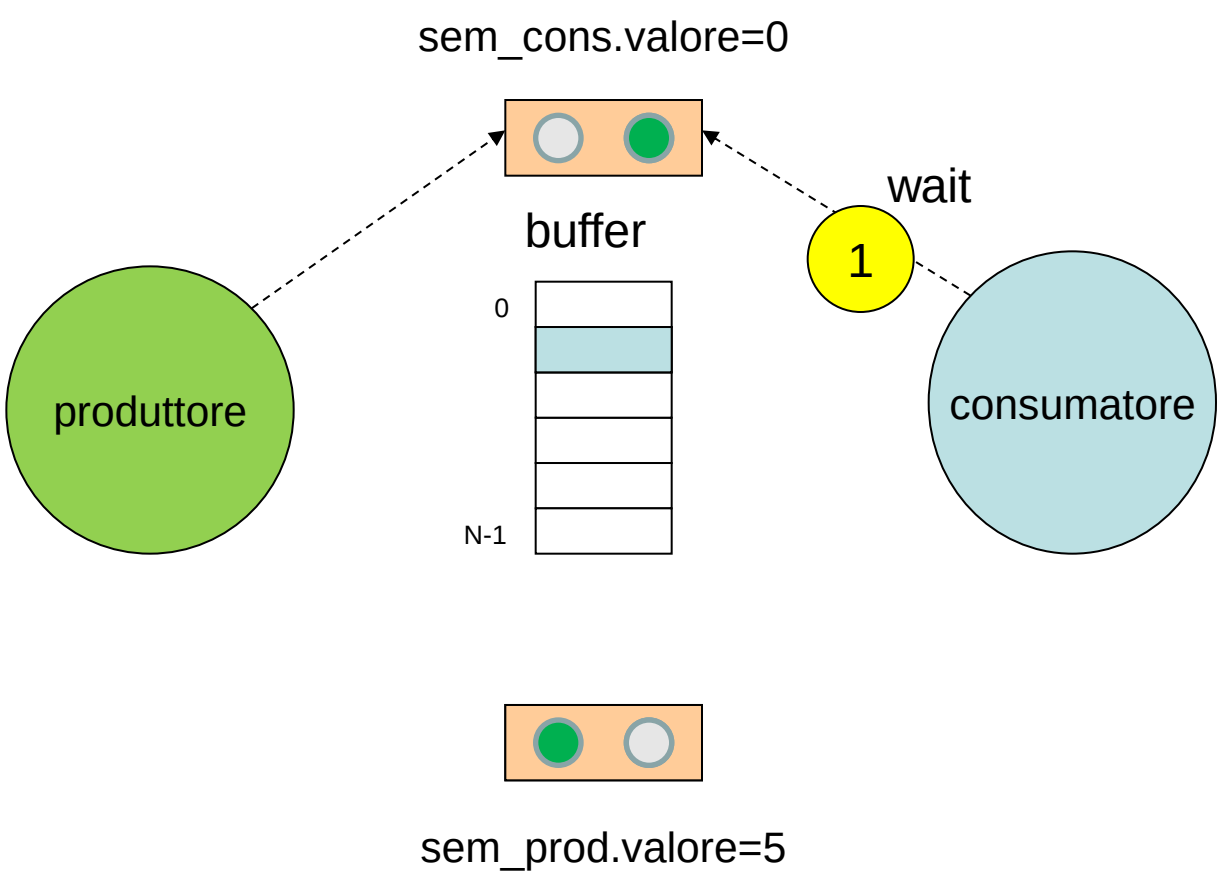


Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

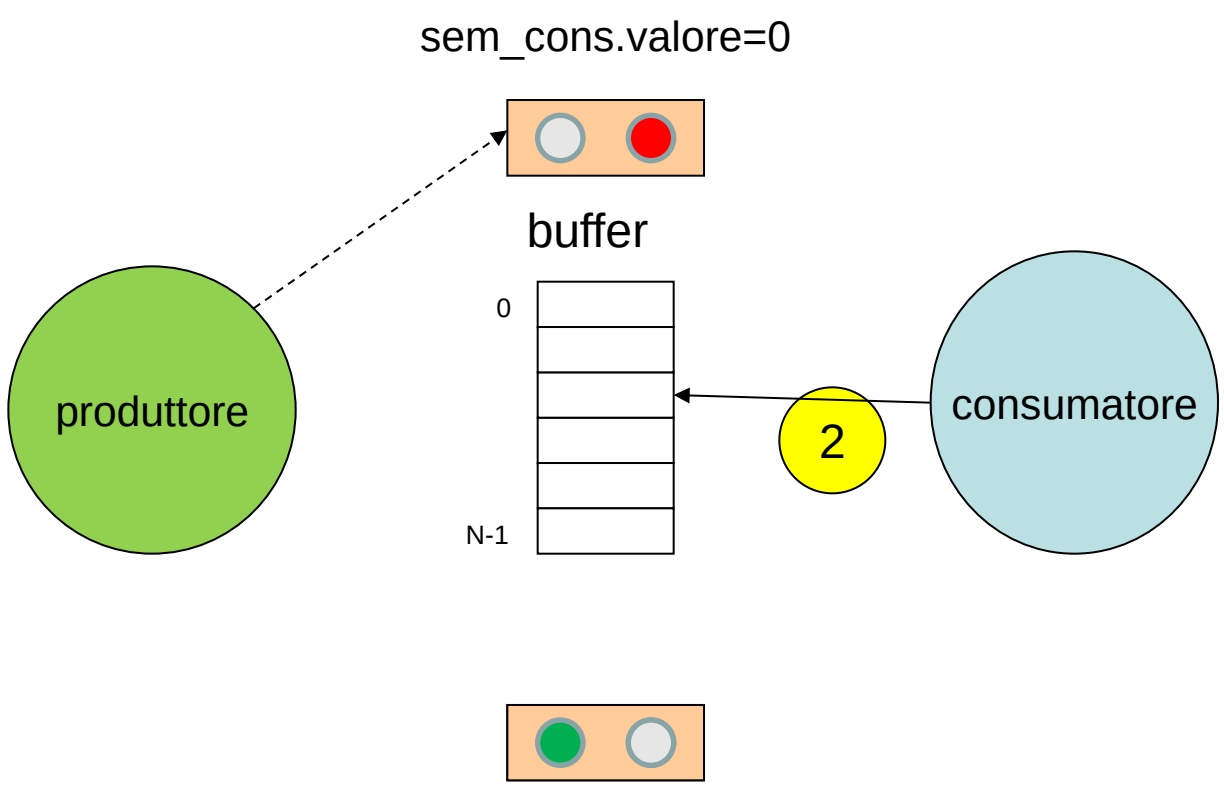


Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

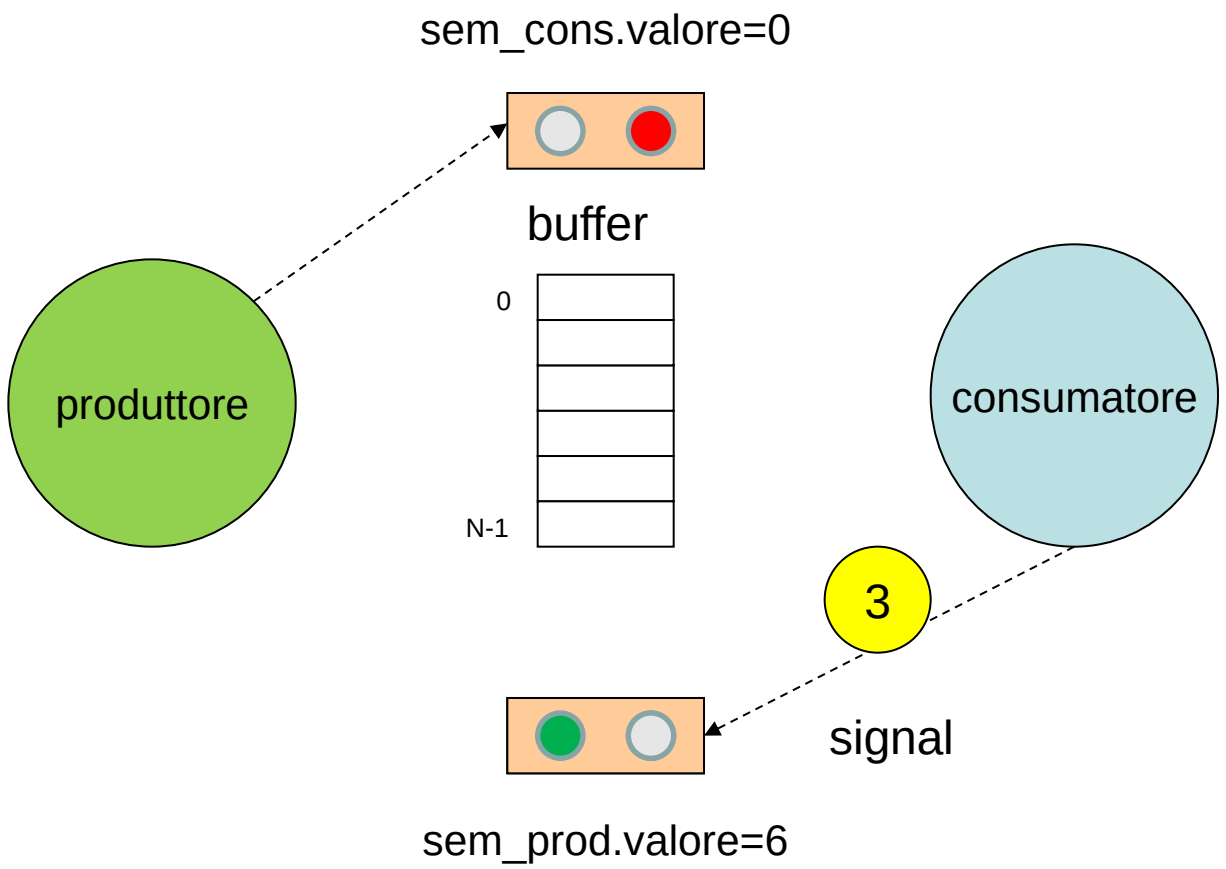


Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica 2

```
x=buffer[leggi];
leggi=(leggi+1)%N
```



Sezione critica

```
buffer[scrivi]=x;
scrivi=(scrivi+1)%N
```

Sezione critica

```
x=buffer[leggi];
leggi=(leggi+1)%N
```

```
produttore (){
  do {
     <produzione del messaggio x>;
     wait (sem_prod);
     buffer[scrivi]=x; // inserimento del messaggio
     scrivi=(scrivi+1)%N;
     signal(sem_cons);
  } while (!fine);
}

consumatore () {
  do {
     wait (sem_cons);
     x=buffer[leggi]; // prelievo del messaggio
     leggi=(leggi+1)%N;
     signal(sem_prod)
     <consumo del messaggio x>
  } while (!fine);
}
```

Interazione tra processi

- I processi possono cooperare tra loro o competere per l'uso di risorse comuni.
- Nei sistemi che seguono il modello ad ambiente locale, un processo ha un proprio spazio di indirizzamento privato e pertanto non può condividere dati con altri processi.
- Nei sistemi POSIX, la sincronizzazione può avvenire attraverso lo scambio di ***segnali****,* mentre la comunicazione può realizzarsi mediante l'uso di memoria condivisa e/o lo scambio di messaggi oppure utilizzando ***pipe*** *e/o* ***socket****.*

Sincronizzazione con segnali

- In POSIX, la sincronizzazione avviene mediante i segnali, meccanismi realizzati a livello di kernel che consentono la notifica di eventi asincroni tra processi.

- Il segnale è un evento che un processo mittente invia ad uno o più processi destinatari. Il segnale genera nel processo destinatario un'interruzione del flusso di esecuzione.
- In particolare, quando un processo riceve un segnale, può comportarsi in uno dei seguenti modi:
  - **Eseguire un'azione predefinita dal sistema operativo**
  - **Ignorare il segnale**
  - **Gestire il segnale con una funzione (handler) definita dal programmatore**
- Diverse implementazioni POSIX possono avere diversi segnali. Ogni segnale è identificato da un numero **intero** e da un **nome simbolico,** definiti nel file header di sistema **signal.h.**
- Con la shell, si può visualizzare l'elenco dei segnali mediante il comando ***kill –l***.

- In particolare, sono disponibili 2 segnali **SIGUSR1** e **SIGUSR2** a cui non è associata nessuna azione di default. Questi segnali possono essere usati dai processi utente per realizzare specifiche politiche di sincronizzazione.
- Alcuni segnali non sono intercettabili mediante handler (ad esempio **SIGKILL** che provoca la terminazione del processo)