Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2018-2019

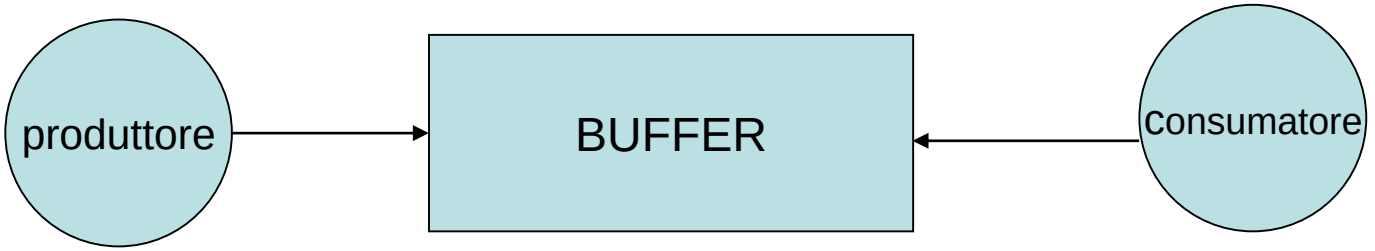
Pietro Frasca

Lezione 10

Giovedì 8-11-2018

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 può ottenere mediante i **semafori**.

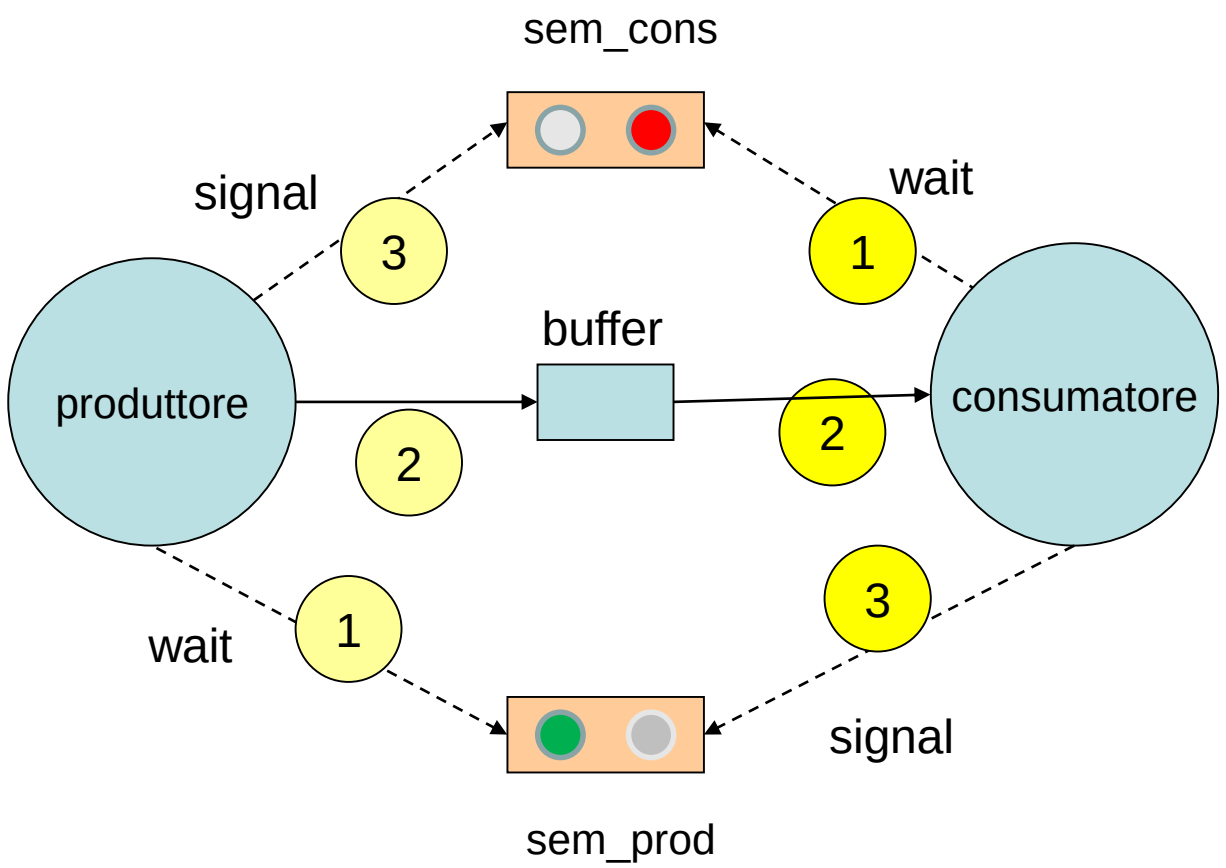


Soluzione al problema della comunicazione con semafori

Soluzione al problema della comunicazione con buffer di capacità 1

Soluzione del problema del produttore-consumatore con buffer di capacità di un messaggio, 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

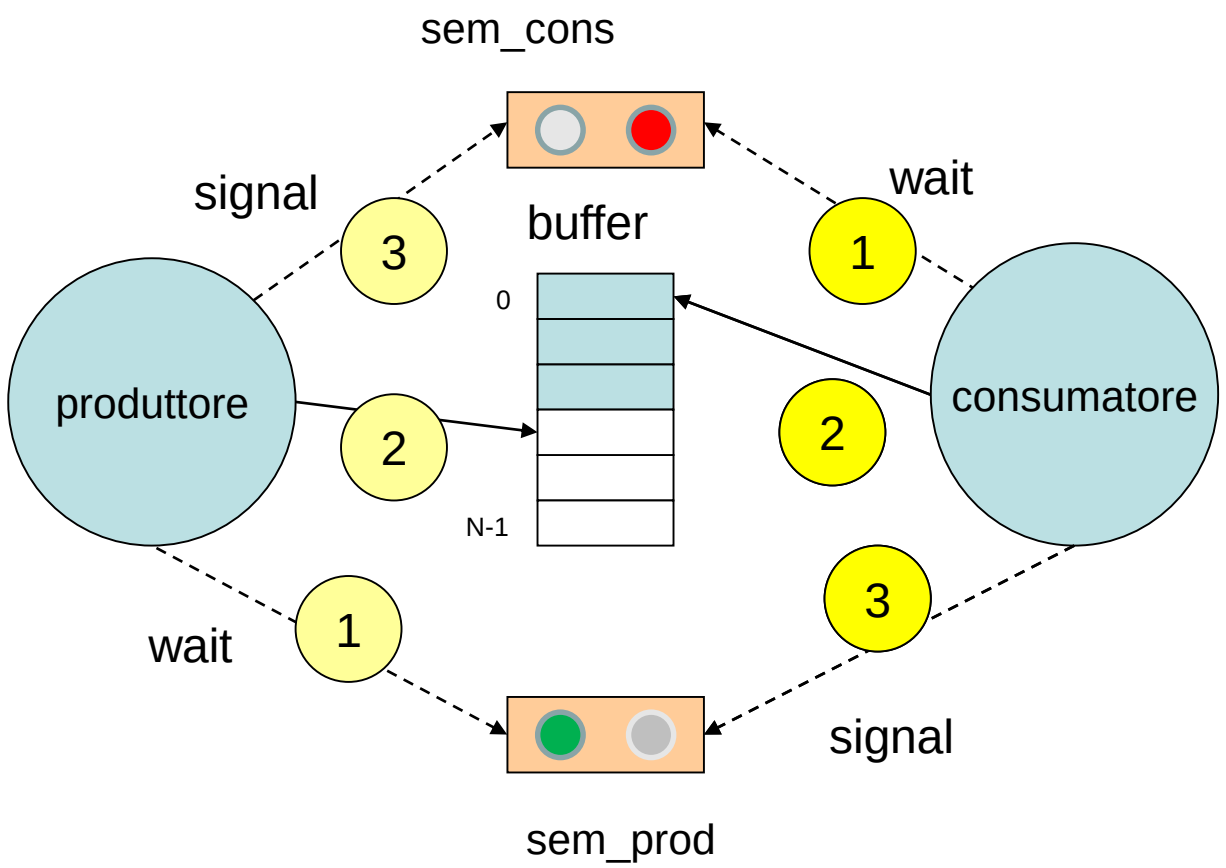
Soluzione del problema del produttore-consumatore con buffer di capacità N (messaggi), utilizzando i semafori è la seguente**:**

- 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;.

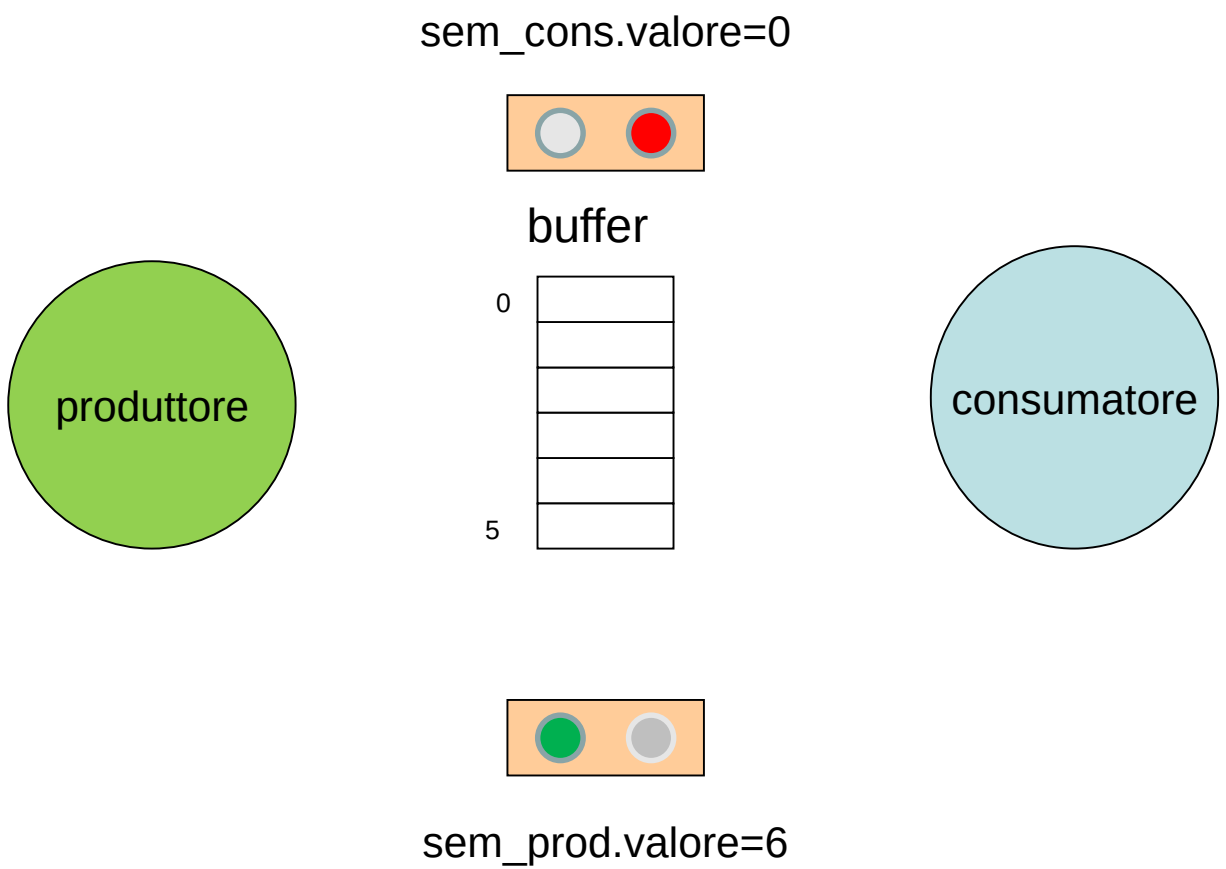


Sezione critica 2

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

Sezione critica 2

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

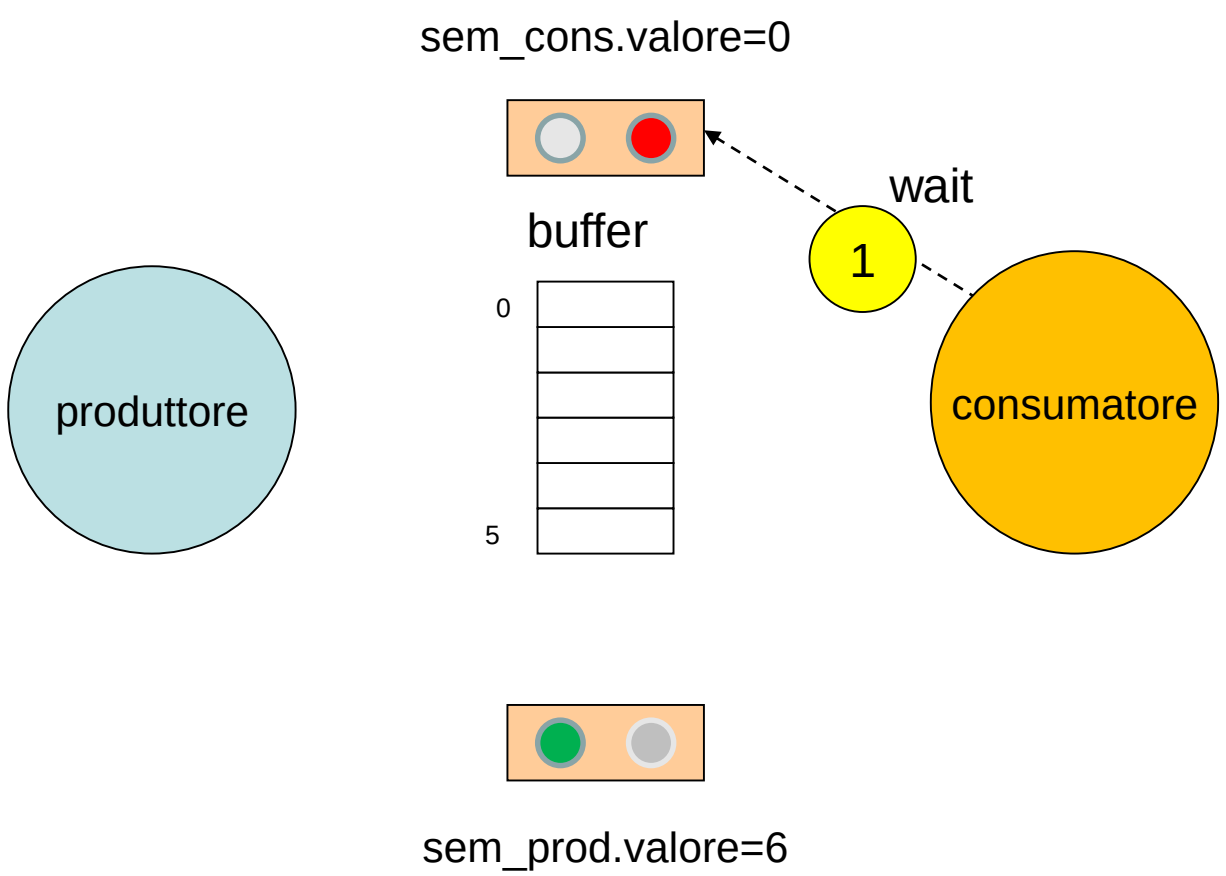


Sezione critica

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

Sezione critica

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

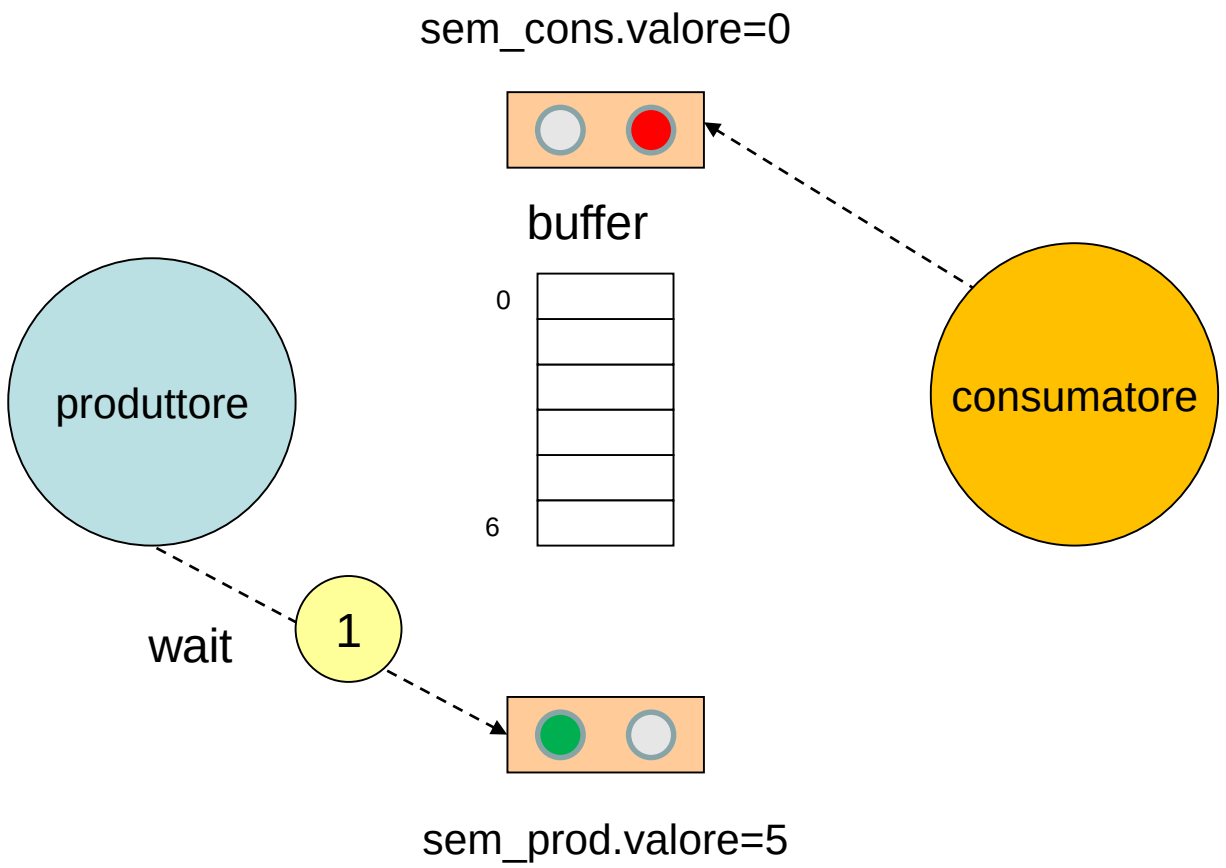


Sezione critica

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

Sezione critica

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

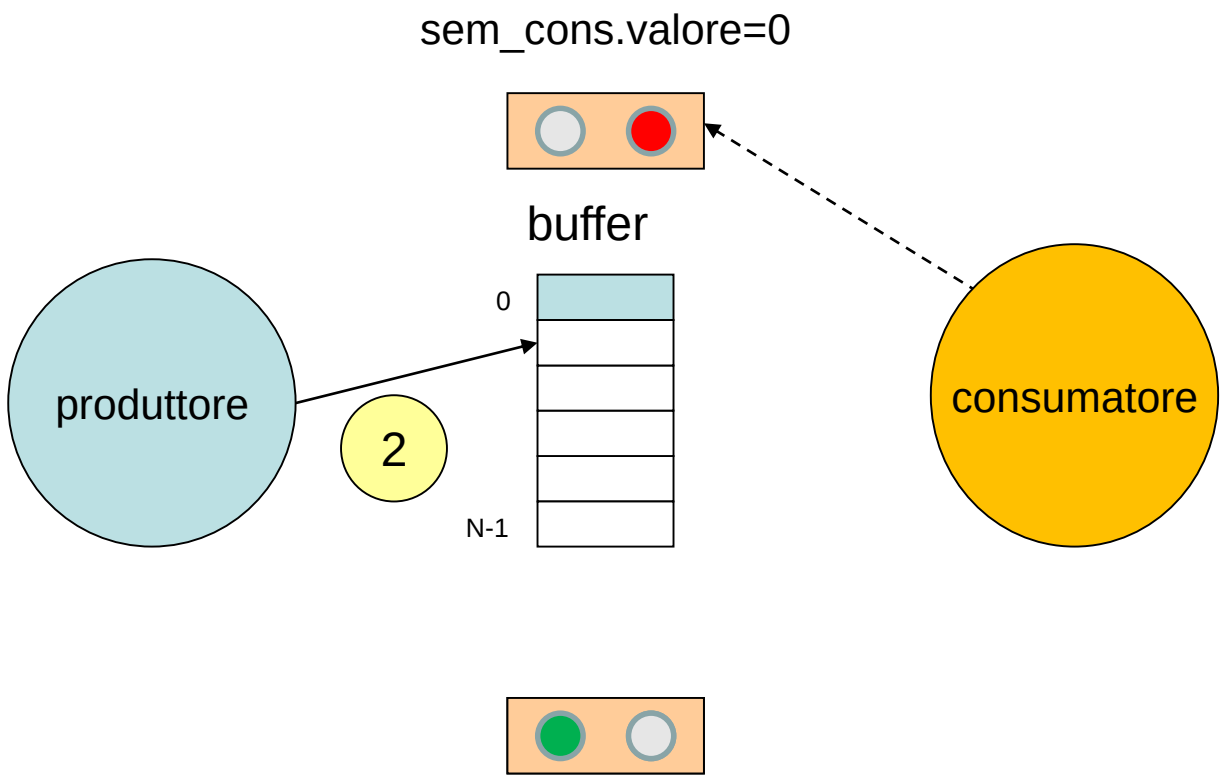


Sezione critica

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

Sezione critica

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

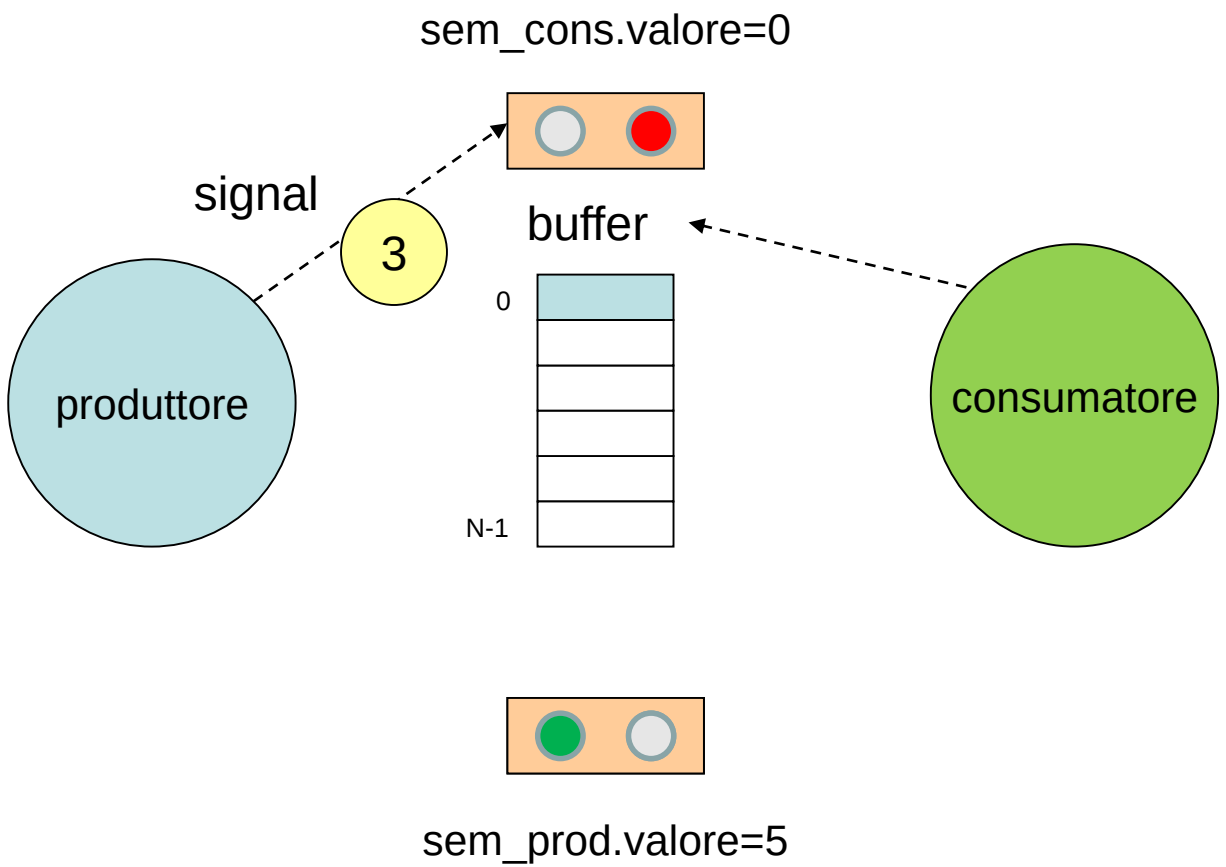


Sezione critica 2

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

Sezione critica

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

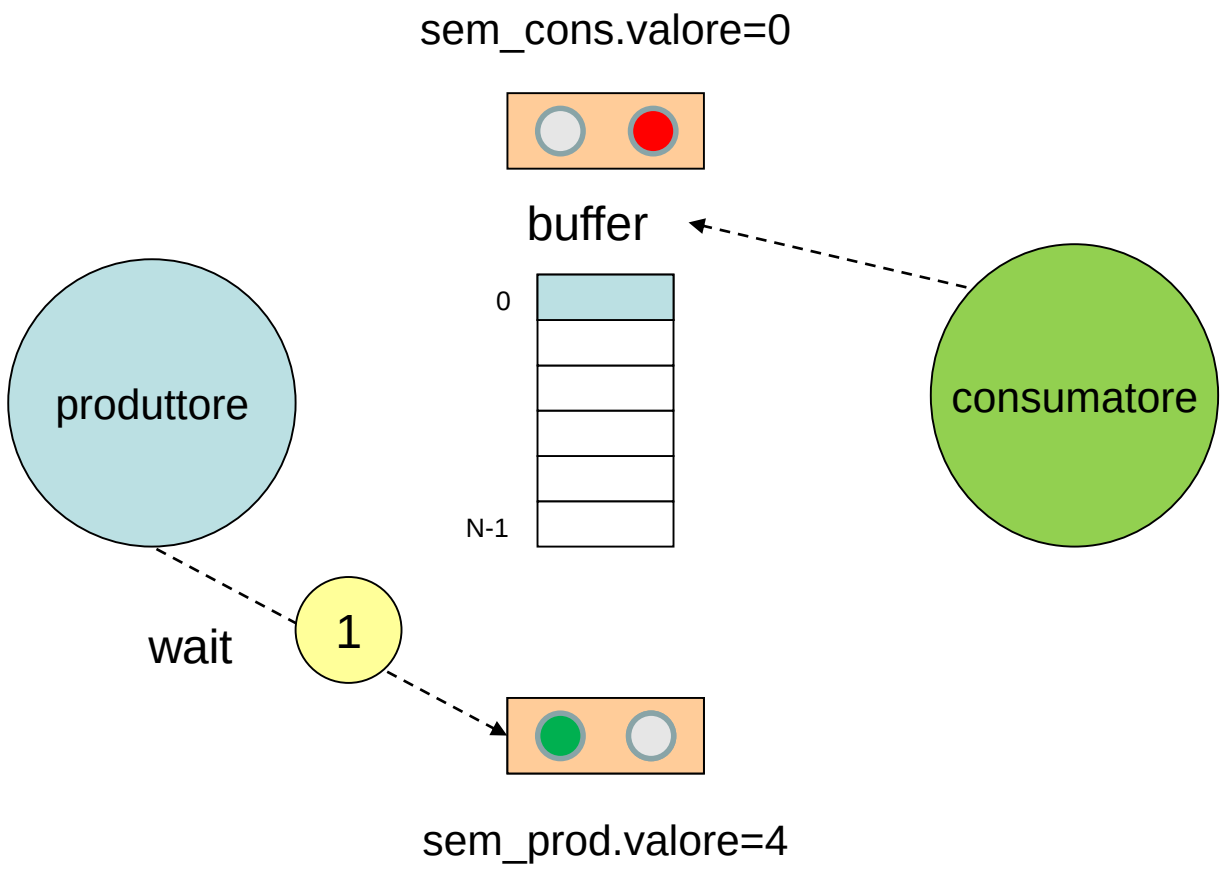


Sezione critica

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

Sezione critica

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

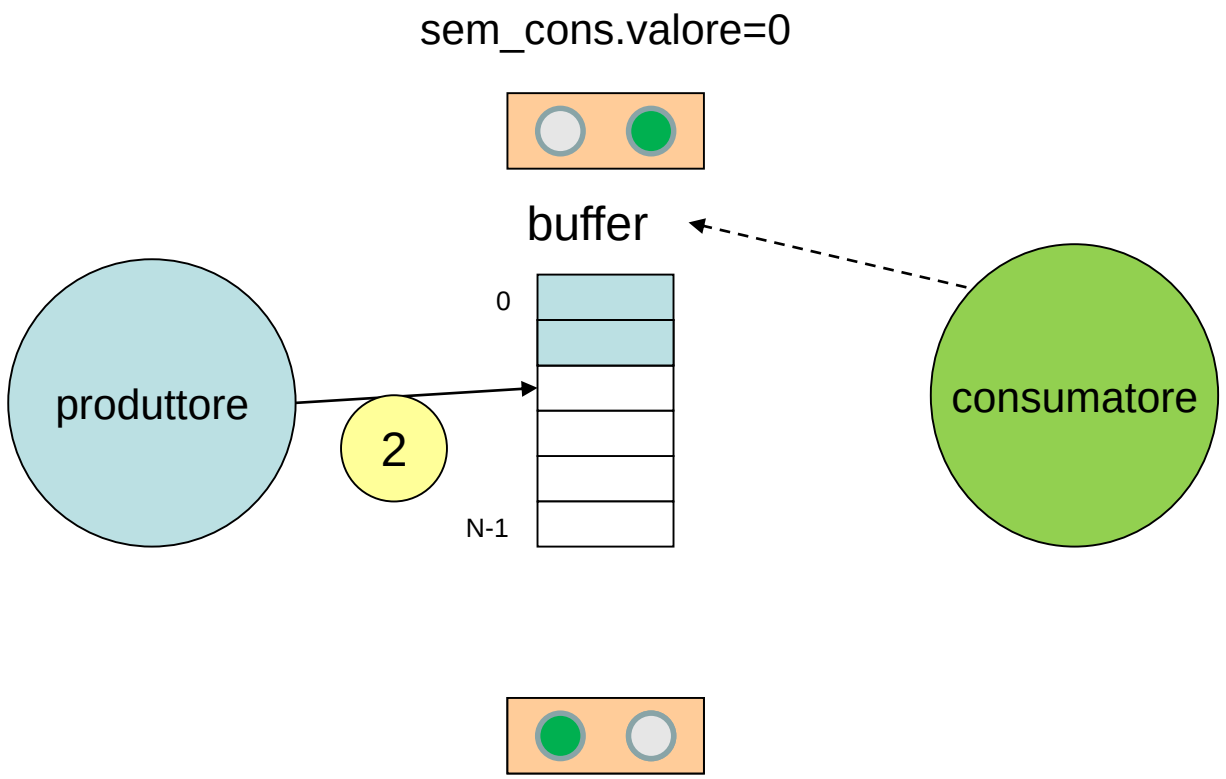


Sezione critica

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

Sezione critica

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

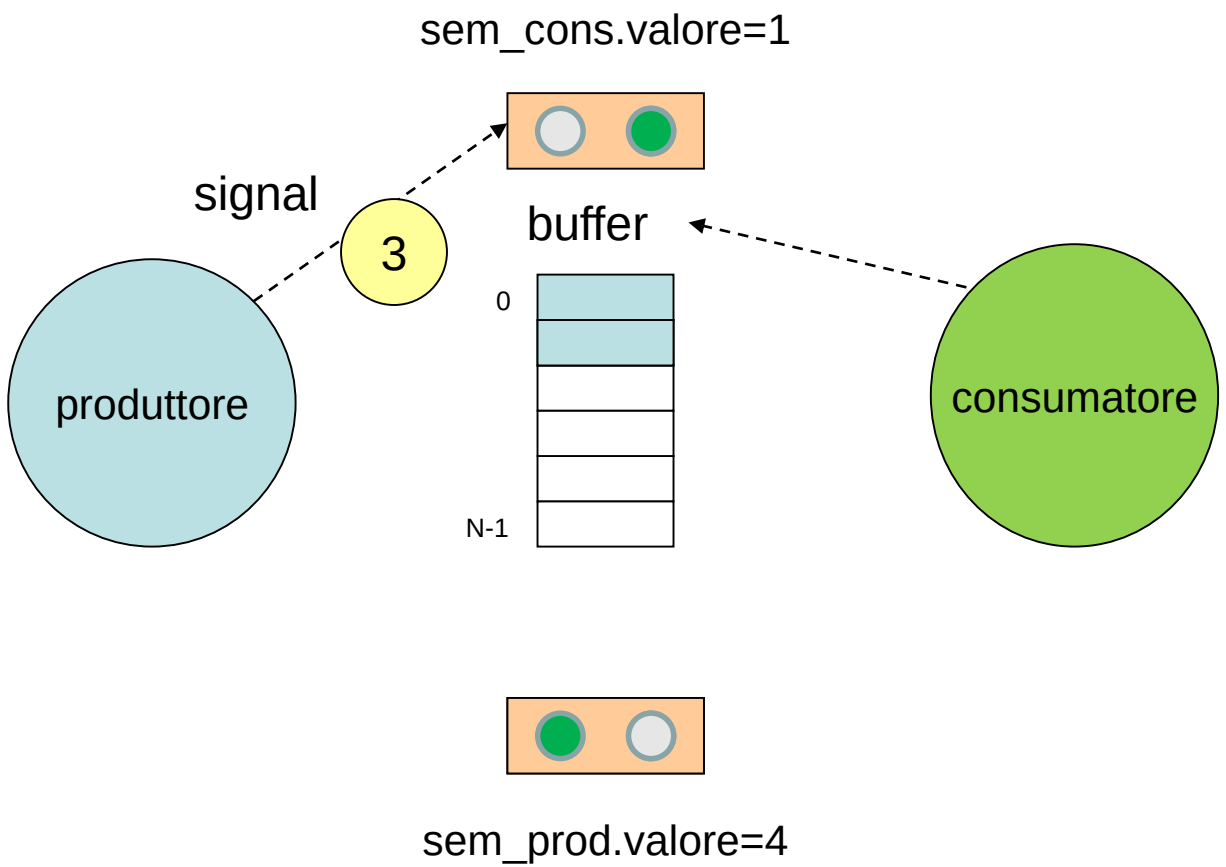


Sezione critica 2

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

Sezione critica

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

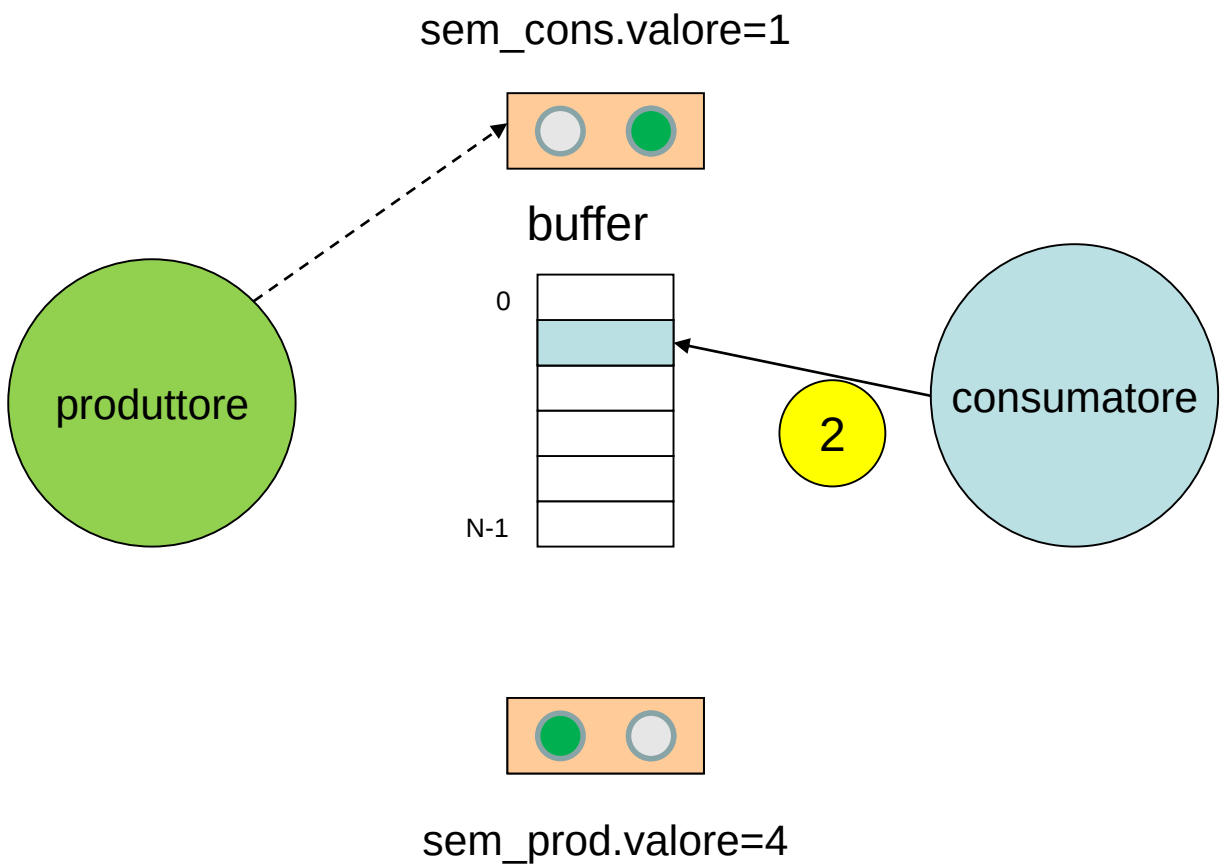


Sezione critica

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

Sezione critica

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

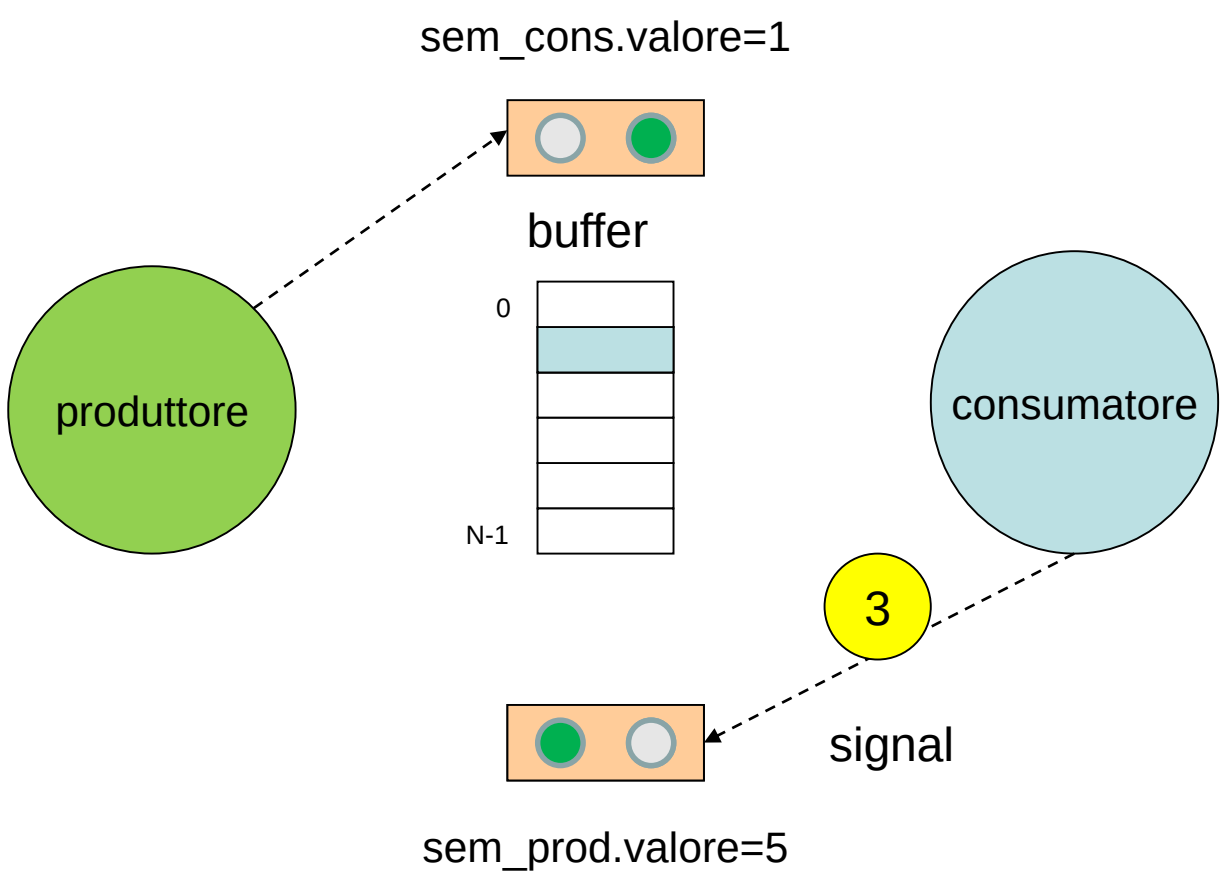


Sezione critica

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

Sezione critica 2

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

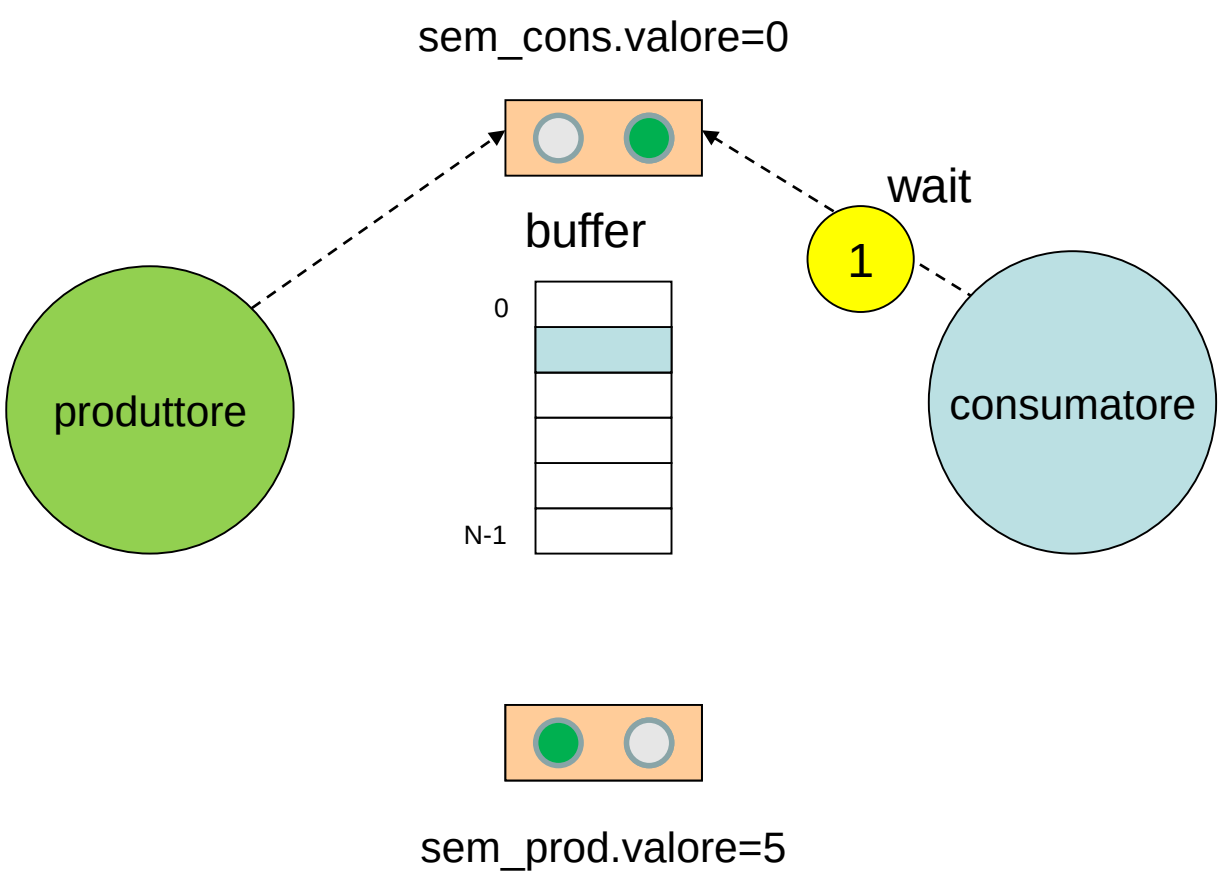


Sezione critica

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

Sezione critica

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

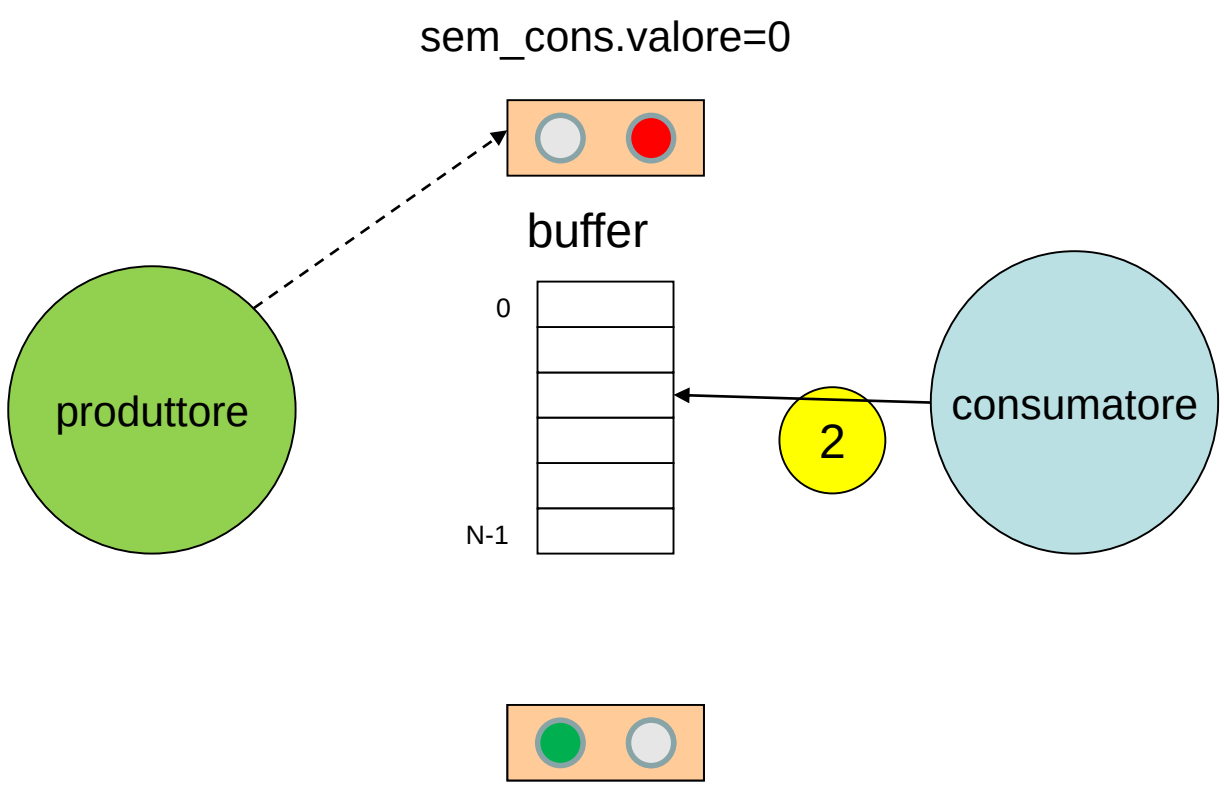


Sezione critica

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

Sezione critica

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

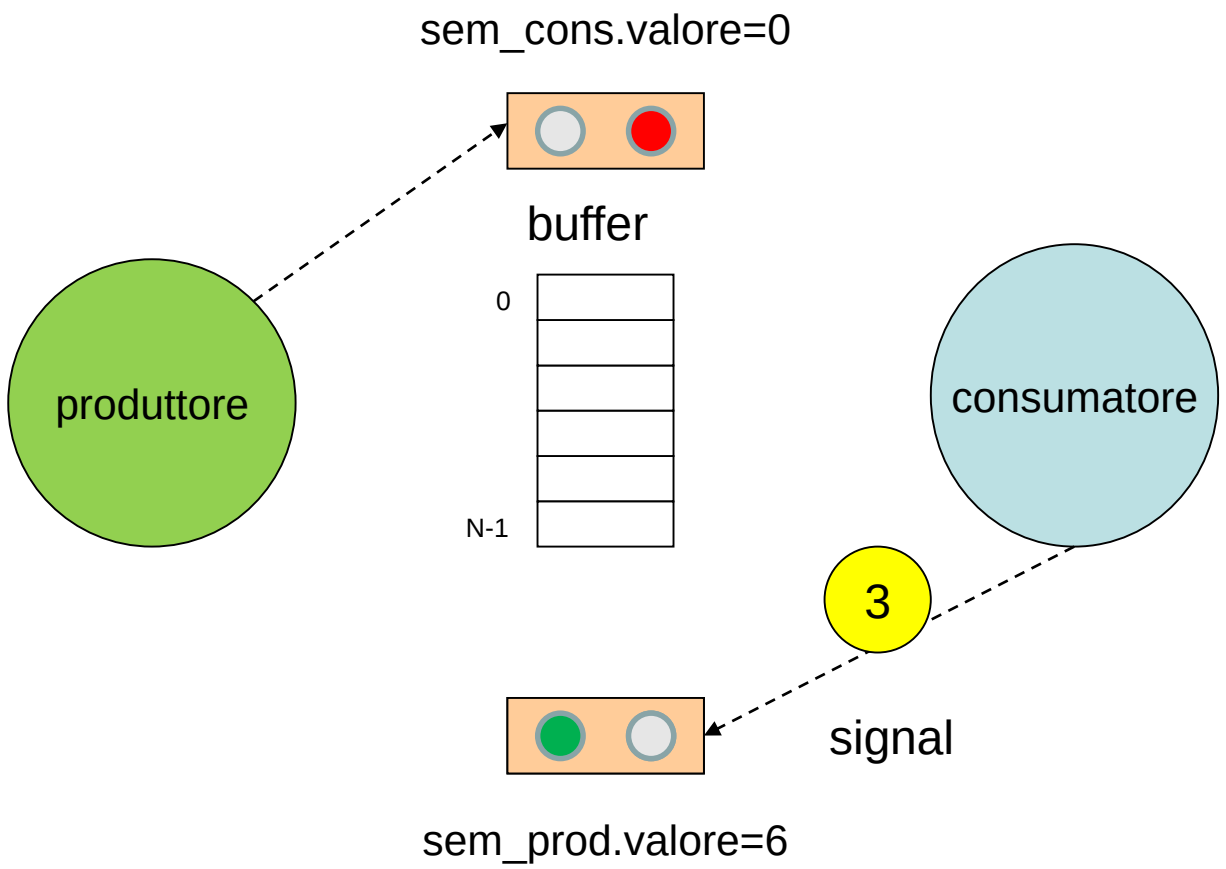


Sezione critica

```
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)

System call per l'uso dei segnali

- Un processo che riceve un segnale può gestire l'azione di risposta alla ricezione di tale evento utilizzando le system call sigaction() o *signal()*.
- La chiamata ***sigaction()*** appartenente allo standard POSIX è più complessa ed è da preferirsi alla *signal(),* definita nello *standard C*.
- Tuttavia, per semplicità, analizzeremo la *signal().*

```
void (*signal(int sig, void (*handler)()))(int);
```

  - ***sig*** è un intero (o la costante simbolica) che specifica il segnale da gestire;
  - ***handler*** è il puntatore alla funzione che implementa il codice da eseguire quando il processo riceve il segnale. Il parametro handler può specificare la funzione di gestione dell'interruzione *(handler),* oppure assumere il valore:

    - ***SIG_IGN*** nel caso in cui il segnale debba essere ignorato;
    - ***SIG_DFL*** nel caso in cui debba essere eseguita l'azione di default.
- la funzione ***handler()*** ha un parametro di tipo intero che, al momento della sua attivazione, assumerà il valore dell'identificativo del segnale che ha ricevuto.
- Le associazioni tra segnali e azioni sono registrate nel PCB del processo.
- Poiché la ***fork()*** copia parte delle informazioni del PCB del padre, comprese quelle riguardanti i segnali e rispettivi handler, nel PCB del figlio, e che padre e figlio condividono lo stesso codice, il figlio eredita dal padre le informazioni relative alla gestione dei segnali e quindi:
  - **Le azioni di default dei segnali del figlio sono le stesse del padre;**
  - **ogni processo figlio ignora i segnali ignorati dal padre;**
  - **ogni processo figlio gestisce i segnali con le stesse funzioni usate dal padre;**

- Dato che padre e figlio hanno *PCB* distinti, eventuali chiamate *sigaction()* o *signal()* eseguite dal figlio sono indipendenti dalla gestione dei segnali del padre.
- Inoltre, un processo quando chiama una funzione della famiglia *exec()* non mantiene l'associazione segnale/handler dato che una *exec()* mantiene parte delle informazioni del PCB del processo che la chiama, ma **non dati e codice** e quindi neanche le funzioni di gestione dei segnali.
- L’esempio seguente mostra l’uso della system call *signal()*.

```
#inc1ude <signal.h>
void gestore (int signum){
  printf("Ricevuto il segnale %d \n", signum);
  /* In alcune versioni di unix l'associazione
   segnale/gestore non è persistente. In questo caso è
   necessario rieseguire la signal.
  */
  // signal(SIGUSR1, gestore);
}
main () (
  signal (SIGUSRl, gestore) ;
  /* da qui in poi il processo eseguirà la funzione gestore
   quando riceverà il segnale SIGUSRl */
  ..............
  signal (SIGUSRl, SIG_IGN) ;
  / * SIGUSRl è da qui ignorato: il processo
      non eseguirà più la funzione gestore in risposta a
       SIGUSRl */
}
```

Invio di segnali tra processi

- I processi possono inviare segnali ad altri processi con la system call ***kill()***:

```
#include <signal.h>
int kill (int pid, int sig);
```

  - ***pid*** è il pid del processo destinatario del segnale ***sig***. Se ***pid*** vale **zero**, il segnale ***sig*** viene inviato a tutti i processi della gerarchia del processo mittente.
  - ***sig*** è il segnale da inviare, espresso come numero o come costante simbolica.
- L'esempio seguente mostra l'uso di *signal()* e *kill()*. Il programma, genera due processi (padre e figlio). Entrambi i processi gestiscono il segnale SIGUSR1 mediante la funzione gestore: il figlio, infatti, eredita l'impostazione della signal del padre chiamata prima della *fork()*. Una volta attivi entrambi i processi, il padre invia continuamente il segnale SIGUSR1 al figlio.

```
#include <signal.h>
void gestore (int signum) {
   static int cont=0;
   printf ("Processo con pid %d; ricevuti n. %d segnali %d
            \n",  getpid(), ++cont, signum);
}
int  main () {
  pid_t pid;
  signal(SIGUSR1, gestore);
  pid = fork ();
  if (pid==0) /* figlio */
    for (; ;) pause();
  else /* padre */
    for ( ; ;) {
      kill (pid, SIGUSR1);
      sleep(1);
     }
}
```

- Oltre alla system call *kill()*, esistono altre chiamate di sistema che automaticamente inviano segnali. Ad esempio la funzione ***alarm()*** causa l'invio del segnale ***SIGALRM*** al processo che la chiama, dopo un intervallo di tempo specificato nell'argomento della funzione.

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds)
```

- L'esempio seguente mostra l'uso di ***alarm()*** e ***pause()***. Dopo **ns** secondi viene inviato un segnale di allarme (SIGALRM) e viene eseguita la funzione ***azione()*** specificata in ***signal()***. Il tempo di allarme ns viene incrementato dopo ogni chiamata di *alarm()*.
- La function ***system()*** manda in esecuzione il programma specificato nell'argomento. Nell'esempio viene eseguita la funzione ***system()*** che consente di mandare in esecuzione un programma specificato nell'argomento, in questo caso viene eseguito l'utility **date** che visualizza data e ora correnti.

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
int ns=1; // periodo iniziale di allarme (1 secondo)
int nmax=10; // valore massimo dell'intervallo di allarme
void azione(){
  /* questa funzione viene eseguita ogni volta
     che il processo riceve il segnale SIGALRM,
  */
  printf("Segnale di allarme ricevuto...eseguo date \n");
  system("date"); // esegue il comando date

  /*
    riassegnamento del periodo di allarme
    che cancella il precedente periodo assegnato.
  */
  alarm(ns); // ns viene incrementato
}
```

```
int main() {
  int i;
  signal(SIGALRM,azione);
  alarm(ns);
  while(ns <= nmax) {
    printf("processo in pausa\n");
    pause();
    printf("fine pausa\n");
    ns++;// incremento del periodo di allarme
   }
   exit(0);
}
```