Problemi di cooperazione nel modello ad ambiente globale

Corso di Laurea in Ingegneria Informatica

Università degli Studi di Napoli Federico II

Anno Accademico 2024/2025, Canale San Giovanni

Problemi di cooperazione nel modello ad ambiente globale

- Sommario
  - Il problema produttore/consumatore
    - Soluzione con buffer singolo
    - Soluzione con coda circolare
    - Soluzione con un pool di buffer e vettore di stato
  - Il problema lettori/scrittori
    - Soluzione con starvation degli scrittori
    - Soluzione con starvation di entrambi
- Riferimenti
  - Dispensa su problemi di programmazione concorrente (Ancillotti – Boari, Principi e Tecniche di Programmazione Concorrente)

Differenze tra problemi di cooperazione e mutua esclusione

- Pur esistendo un problema (potenziale) di mutua esclusione nell'utilizzo di buffer comuni...
- ...la soluzione impone un **ordinamento** nelle operazioni dei processi

È necessario che i processi si coordinino, indicando agli altri quando le loro operazioni sono avvenute

Il problema produttore/consumatore

Due categorie di processi:

- **Produttori**, che depositano un messaggio su di una risorsa condivisa
- **Consumatori**, che prelevano il messaggio dalla risorsa condivisa

Vincoli:

- Il produttore **non può produrre** un messaggio prima che qualche consumatore abbia **prelevato** il messaggio precedente
- Il consumatore **non può prelevare** alcun messaggio fino a che un produttore non l'abbia **depositato**

Prod-Cons con un unico buffer

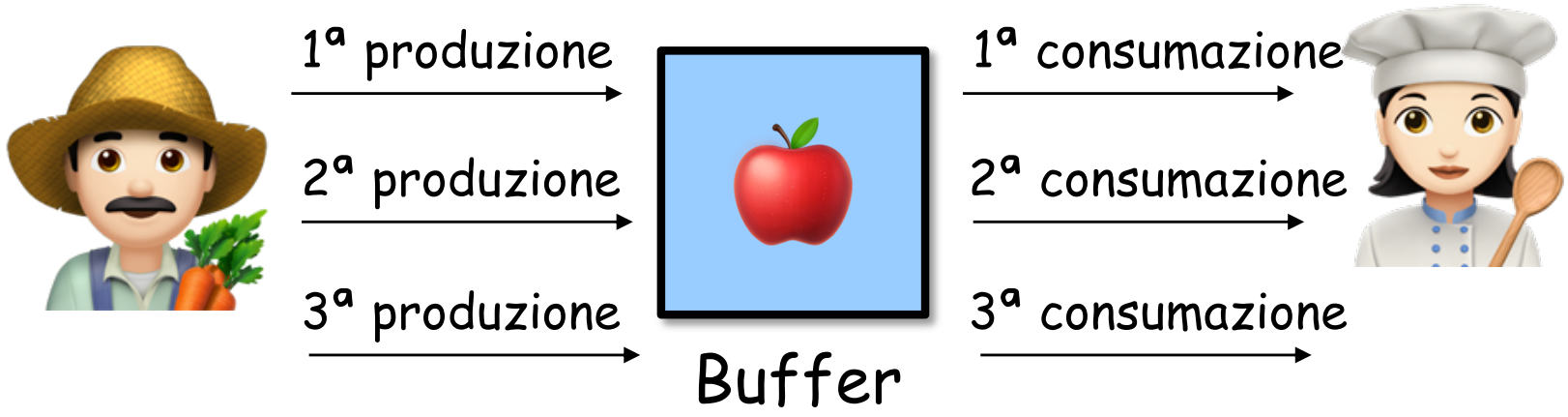
Problema dei Produttori/Consumatori attraverso l'utilizzo di un **buffer singolo**

Prod-Cons con un unico buffer

I vincoli imposti da un problema di tipo Produttore-Consumatore, nel caso che il **buffer sia unico** e che vi siano **un solo produttore** ed **un solo consumatore**, sono i seguenti:

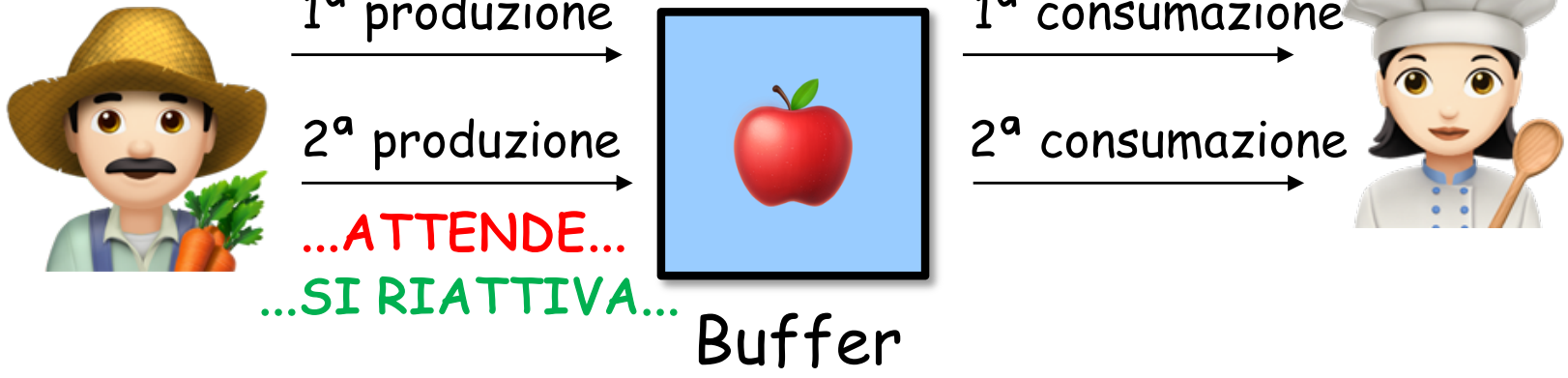
- Il produttore non può produrre un messaggio se il consumatore non ha consumato il messaggio precedente
- Il consumatore non può prelevare un messaggio se prima il produttore non l'ha depositato

Prod-Cons con un unico buffer



È necessario che produttori e consumatori si coordinino per indicare rispettivamente l'avvenuto deposito e prelievo

Prod-Cons con un unico buffer



Il produttore non può produrre un messaggio se il consumatore non ha consumato il messaggio precedente

Prod-Cons con un unico buffer

Per la sincronizzazione dei processi produttore e consumatore si utilizzano **due semafori**:

1. **SPAZIO_DISP**: semaforo bloccato da un produttore prima di una produzione, e sbloccato da un consumatore in seguito ad un consumo (**VALORE INIZIALE: 1**)

2. **MSG_DISP**: semaforo sbloccato da un produttore in seguito ad una produzione, e bloccato da un consumatore prima del consumo (**VALORE INIZIALE: 0**)

Prod-Cons con un unico buffer

La produzione ed il consumo avvengono rispettivamente all'interno delle procedure

void Produttore(msg*, int)
void Consumatore(msg *, int)

all'interno delle quali si effettuano anche le operazioni di **Wait_Sem** e **Signal_Sem** necessarie per la sincronizzazione

Prod-Cons con un unico buffer

```
void Produttore(msg * ptr_sh, int sem){

        msg mess;

        Wait_Sem(sem, SPAZIO_DISP);

        // Produzione di valore_prodotto ...
        printf ("Produzione in corso…");
        mess = valore_prodotto;

        *ptr_sh = mess;

        Signal_Sem(sem, MSG_DISP);
}
```

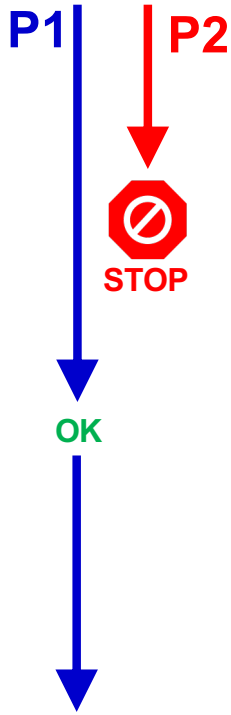
Prod-Cons con un unico buffer

```
void Produttore(msg * ptr_sh, int sem){

        msg mess;

        Wait_Sem(sem, SPAZIO_DISP);

        // Produzione di valore_prodotto ...
        printf ("Produzione in corso…");
        mess = valore_prodotto;

        *ptr_sh = mess;

        Signal_Sem(sem, MSG_DISP);
}
```

La Wait_Sem():

- **sospende** il chiamante se `SPAZIO_DISP == 0`
- **non sospende** il chiamante se `SPAZIO_DISP==1`, pone `SPAZIO_DISP=0`

Prod-Cons con un unico buffer




```
void Produttore(msg * ptr_sh, int sem){
    msg mess;

    Wait_Sem(sem, SPAZIO_DISP);

    // Produzione di valore_prodotto ...
    printf ("Produzione in corso…");
    mess = valore_prodotto;

    *ptr_sh = mess;

    Signal_Sem(sem, MSG_DISP);
}
```

Supponiamo che inizialmente il buffer sia **vuoto**
`SPAZIO_DISP = 1`
`MSG_DISP = 0`

Alla **1ª produzione**, il processo **entra nella sez. critica**, pone `SPAZIO_DISP = 0` (s.value--)

All'**uscita**, pone `MSG_DISP = 1`

Alla **2ª produzione**, il processo **viene sospeso**, finché non si ha `SPAZIO_DISP = 1` (per effetto di una **signal** dal consumatore)

Prod-Cons con un unico buffer

```
void Consumatore(msg * ptr_sh, int sem){

        msg mess;

        Wait_Sem(sem, MSG_DISP);

        // Prelievo del messaggio
        mess = *ptr_sh;
        printf("Messaggio letto: <%d> \n", mess);

        Signal_Sem(sem, SPAZIO_DISP);

}
```

Prod-Cons con un unico buffer

```
void Consumatore(msg * ptr_sh, int sem){
C1
        msg mess;

        Wait_Sem(sem, MSG_DISP);

        // Prelievo del messaggio
        mess = *ptr_sh;
OK      printf("Messaggio letto: <%d> \n", mess);

        Signal_Sem(sem, SPAZIO_DISP);

}
```

Supponiamo che il produttore abbia già depositato il valore, MSG_DISP = 1

Il consumatore procede a consumare, pone MSG_DISP = 0 (s.value--)

All'uscita, pone SPAZIO_DISP = 1 (s.value++), attiva un produttore in attesa

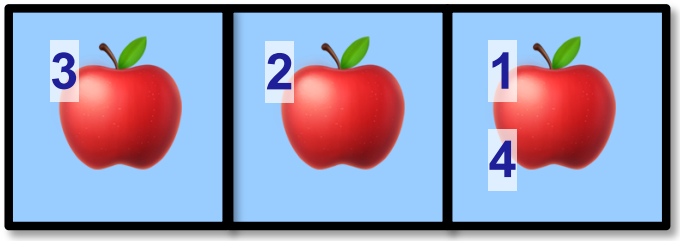
Prod-Cons con una coda

Problema dei Produttori/Consumatori attraverso un **vettore di buffer** (gestito come una **coda circolare**)

Prod-Cons con una coda

1ª prod.
2ª prod.
3ª prod.
4ª prod.

...ATTENDE...
...SI RIATTIVA...



Vettore di Buffer

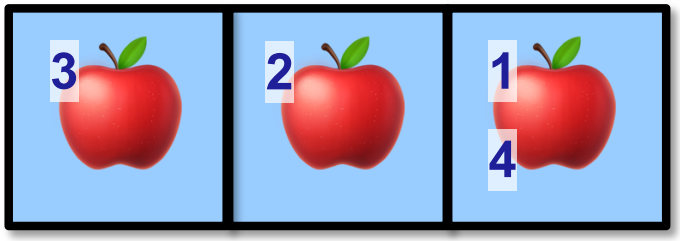
1ª cons.
2ª cons.
3ª cons.
4ª cons.

Il produttore si sospende se i buffer sono tutti pieni.
Il consumatore si sospende se i buffer sono tutti vuoti.
Le operazioni sono svolte in ordine circolare.

Prod-Cons con una coda

1ª prod.
2ª prod.
3ª prod.
4ª prod.

...ATTENDE...
...SI RIATTIVA...



Vettore di Buffer

1ª cons.
2ª cons.
3ª cons.
4ª cons.

Nota: è sufficiente che anche un solo buffer sia disponibile per riprendere le produzioni!

Prod-Cons con una coda

La coda è implementata mediante i seguenti campi:

- **buffer[DIM]** – array di elementi di tipo msg (tipo del messaggio depositato dai produttori) contenente i valori prodotti
- **testa** - tipo intero. Indica la posizione del primo buffer libero in testa, **buffer[testa]**, ossia il primo buffer disponibile per la memorizzazione di un messaggio. L'elemento prodotto più recentemente è alla posizione **buffer[testa-1]**
- **coda** – tipo intero. Indica la posizione dell'elemento prodotto meno recentemente, **buffer[coda]**, da accedere alla prossima consumazione
- Anche le variabili **testa** e **coda** devono essere condivise!

Prod-Cons con una coda

- Per la sincronizzazione dei processi sono stati utilizzati due semafori,
  - **SPAZIO_DISP** indica la presenza di spazio disponibile in coda per la produzione di un messaggio
  - **NUM_MESS** indica il numero di messaggi presenti in coda.
- SPAZIO_DISP ha valore iniziale DIM (dimensione della coda)
- NUM_MESS ha valore iniziale 0

Prod-Cons con una coda

```
void Produttore(int* testa, msg* buffer, int sem) {

        Wait_Sem(sem, SPAZIO_DISP);

        // Produzione di valore_prodotto . . .
        buffer[testa] = valore_prodotto;
        testa = (testa + 1) % DIM;     //gestione circolare

        Signal_Sem(sem, NUM_MESS);       //nelem=nelem+1

}
```

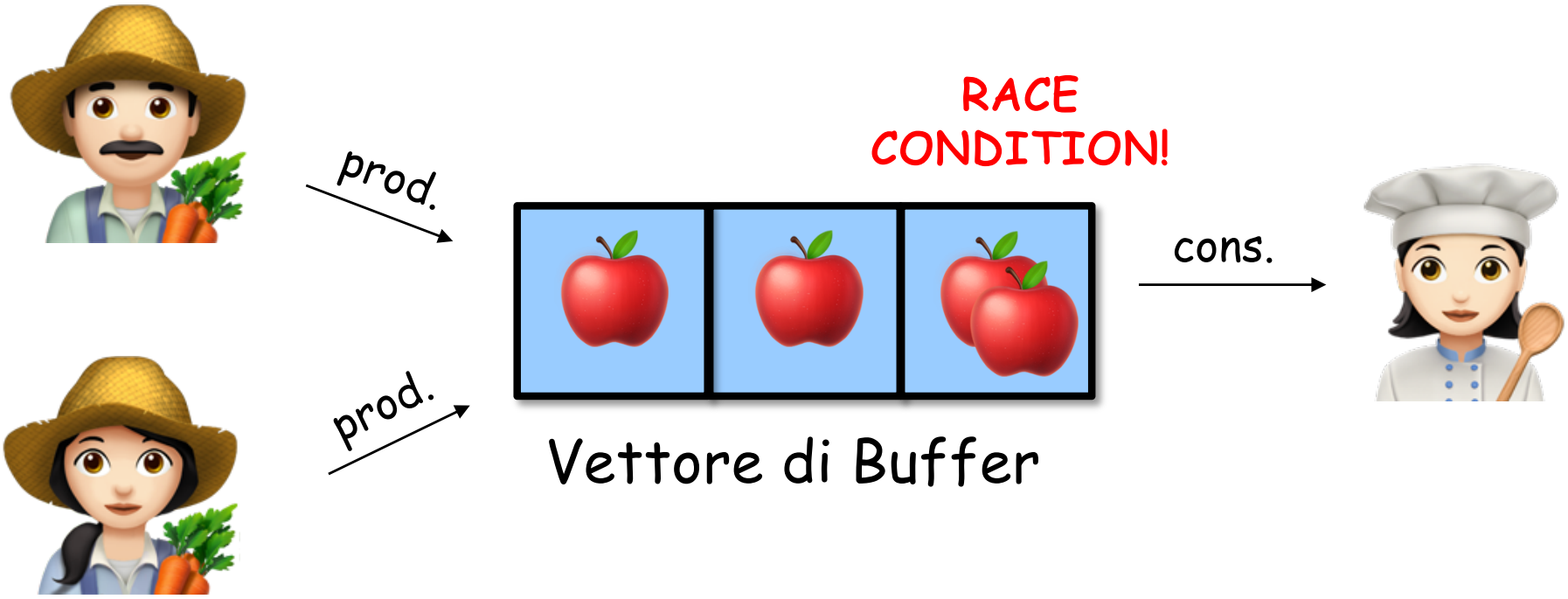
Prod-Cons con una coda

```
void Consumatore(int* coda, msg* buffer, int sem){

        msg mess;

        Wait_Sem(sem, NUM_MESS);

        // Consumo
        mess = buffer[coda];
        coda = (coda + 1) % DIM;   // gestione circolare
        printf("Messaggio letto: <%d> \n", mess);

        Signal_Sem(sem, SPAZIO_DISP);   // nelem=nelem-1
 }
```

Prod-Cons con una coda

- Con questa soluzione il produttore e il consumatore non accedono mai contemporaneamente alla stessa posizione della coda dei messaggi...
- ... tuttavia va bene solo nel caso in cui si ha 1 processo produttore e 1 processo consumatore!

Prod-Cons con una coda



Se vi sono due buffer liberi e due produttori iniziano contemporaneamente a produrre, si verifica una race condition.

Più Produttori e più Consumatori con una coda

- Nell'ipotesi in cui vi siano **più produttori** e **più consumatori** che accedono allo stesso buffer, le operazioni di deposito e prelievo devono essere eseguite rispettivamente in **mutua esclusione**, ed essere quindi programmate come sezioni critiche.

- A tal fine, bisogna introdurre due nuovi semafori:
  - **MUTEX_C** per le operazioni di consumo
  - **MUTEX_P** per le operazioni di produzione
  - Entrambi inizializzati a 1

Più Produttori e più Consumatori con una coda

```
void Produttore(int* testa, msg* buffer, int sem) {

        Wait_Sem(sem, SPAZIO_DISP);
        Wait_Sem(sem, MUTEX_P);     // inizio sez. critica

        // Produzione di valore_prodotto . . .
        buffer[testa] = valore_prodotto;
        testa = (testa + 1) % DIM;  // gestione circolare

        Signal_Sem(sem, MUTEX_P);  // fine sez. critica
        Signal_Sem(sem, NUM_MESS);       // nelem=nelem+1
}
```

Più Produttori e più Consumatori con una coda

```
void Consumatore(int* coda, msg* buffer, int sem) {
        msg mess;

        Wait_Sem(sem, NUM_MESS);
        Wait_Sem(sem, MUTEX_C);     // inizio sez. critica

        // Consumo
        mess = buffer[coda];
        coda = (coda + 1) % DIM;    // gestione circolare
        printf("Messaggio letto: <%d> \n",mess);

        Signal_Sem(sem, MUTEX_C);  // fine sez. critica
        Signal_Sem(sem, SPAZIO_DISP);  // nelem=nelem-1
 }
```

Più Produttori e più Consumatori con una coda

- Per una corretta sincronizzazione, è necessario che gli indici **testa** e **coda** siano condivisi tra i processi
  - es. quando un processo incrementa testa, un altro processo scriverà alla posizione successiva
- È possibile collocare **sia il vettore di buffer sia gli indici testa/coda** nella **stessa memoria condivisa**

```
typedef struct {
    int testa;
    int coda;
    int buffer[DIM];
} prod_cons;

int id_shm = shmget(..., sizeof(prod_cons), ...);
prod_cons * p = shmat(...);
```

Prod-Cons con un pool di buffer

- La soluzione precedente **penalizza** produttori o consumatori "**veloci**" in presenza di produttori o consumatori "**lenti**"
- Può accadere, ad esempio, quando i messaggi prodotti hanno dimensione variabile

Prod-Cons con un pool di buffer

Soluzione: utilizzo di un pool di buffer gestito mediante un **vettore ausiliario di STATO**

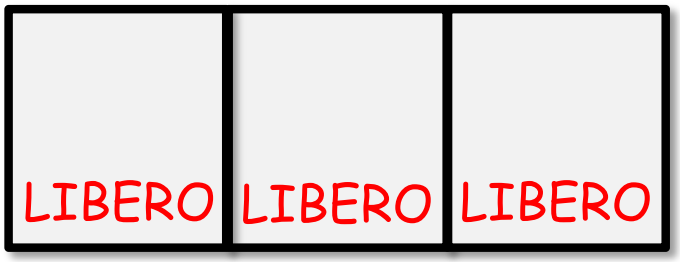
- L'accesso al vettore di stato (per acquisire un buffer) è in **mutua esclusione**
- Dopo aver acquisito un buffer, produttori e consumatori **procedono in concorrenza**

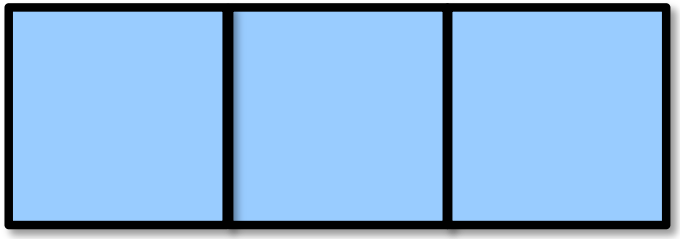
Prod-Cons con un pool di buffer

prod.

prod.

Vettore di stato



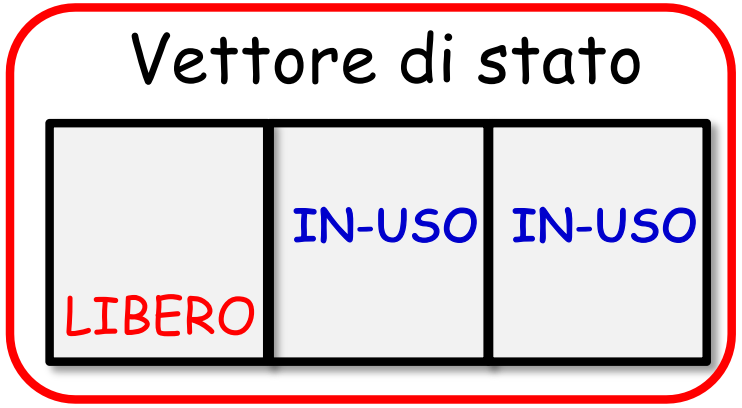


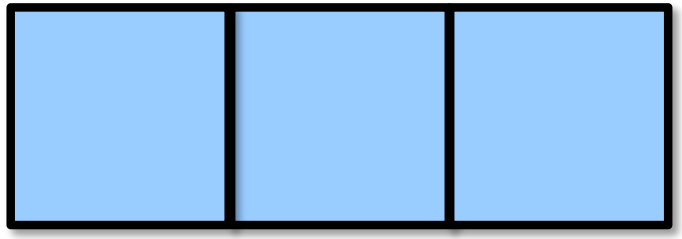
Pool di Buffer

Prod-Cons con un pool di buffer

prod.

prod.



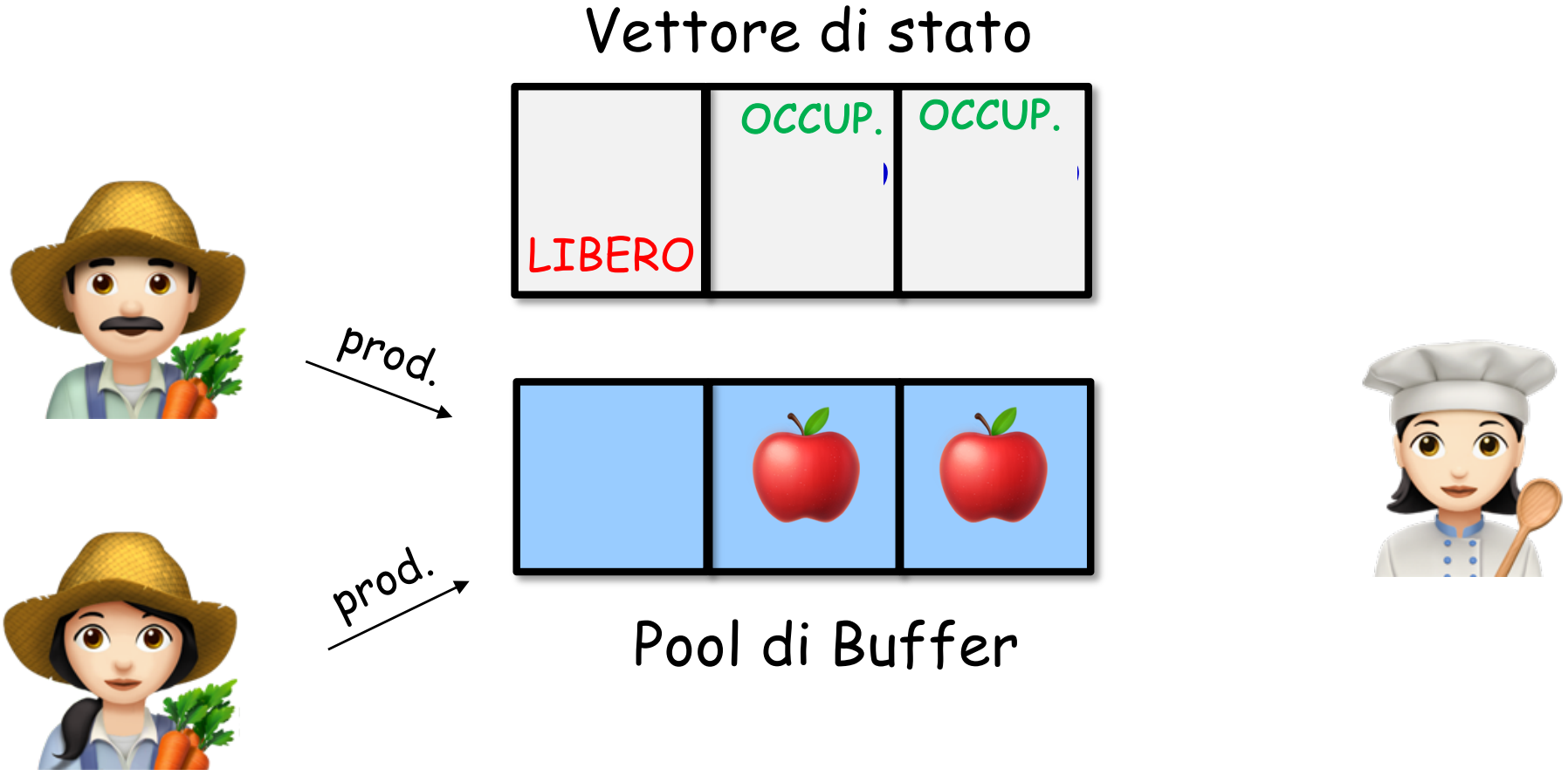


Pool di Buffer

MUTUA ESCLUSIONE

La mutua esclusione è limitata **solo al vettore di stato** (è una operazione veloce).

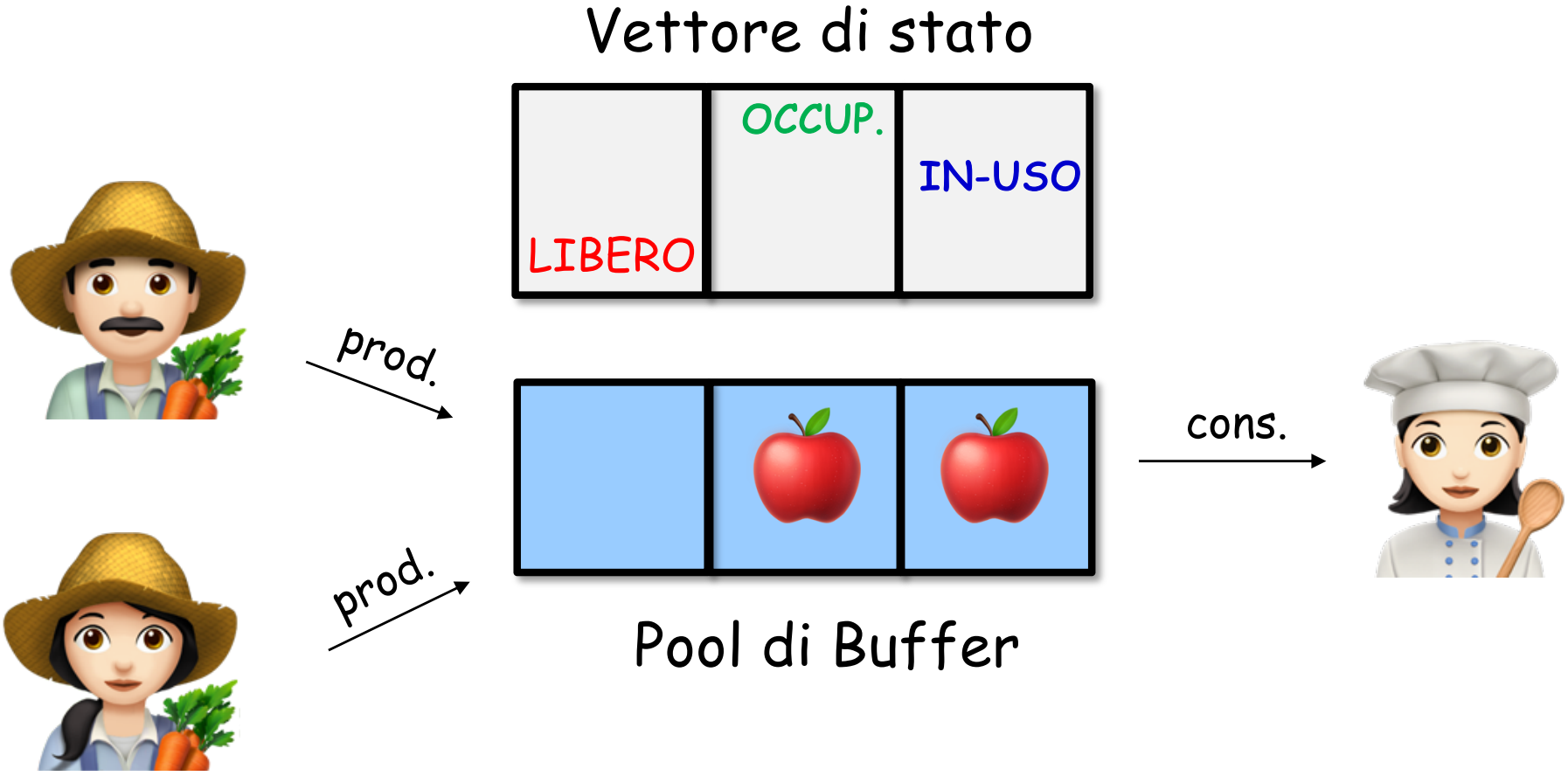
Prod-Cons con un pool di buffer



Il pool di buffer **non è acceduto in mutua esclusione**.

I processi lavorano su **buffer diversi**.

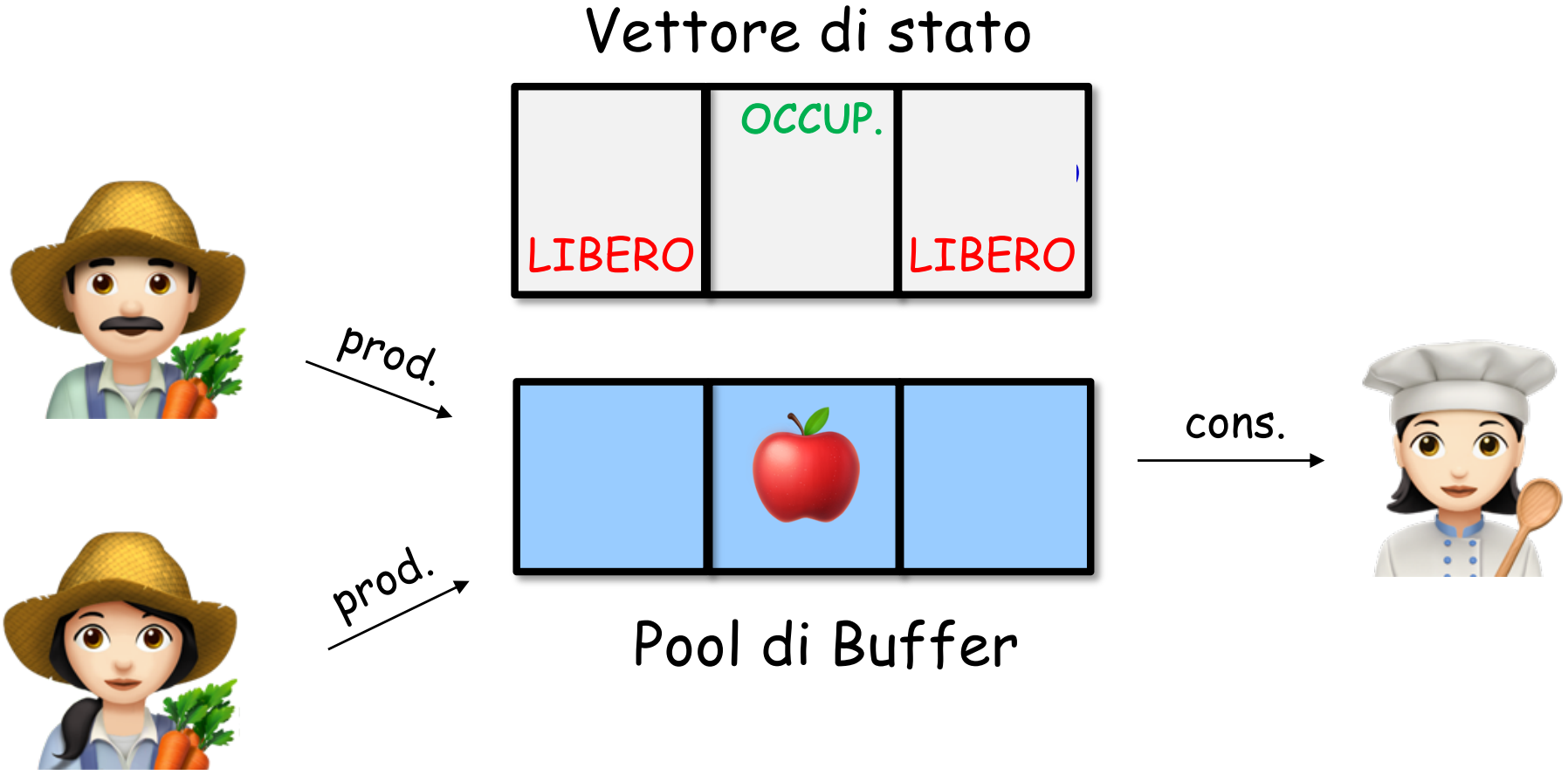
Prod-Cons con un pool di buffer



Il pool di buffer **non è acceduto in mutua esclusione**.

I processi lavorano su **buffer diversi**.

Prod-Cons con un pool di buffer



Il pool di buffer **non è acceduto in mutua esclusione**.

I processi lavorano su **buffer diversi**.

Prod-Cons con un pool di buffer

- Un processo **"lento"** non penalizza i processi **"veloci"**
- L'ordine **non è più circolare**, dipende dalla velocità dei processi

Esempio:

Se il 2° produttore finisce prima, il consumatore preleva dal 2° buffer invece che dal 1° buffer

Prod-Cons con un pool di buffer

La gestione del pool di buffer avviene mediante due vettori:

- **buffer[DIM]** – array di elementi di tipo msg (tipo del messaggio depositato dai produttori) contenente i valori prodotti;

- **stato[DIM]** - array di elementi di tipo intero. Il valore i-simo, stato[i], può assumere i seguenti tre valori:

  - VUOTO – la cella buffer[i] non contiene alcun valore prodotto;
  - PIENO – la cella buffer[i] contiene un valore prodotto e non ancora consumato;
  - IN_USO – il valore della cella buffer[i] contiene un valore in uso da un processo attivo, consumatore o produttore.

Prod-Cons con un pool di buffer

Per la mutua esclusione al buffer di stato, si utilizzano due mutex, **MUTEXP** e **MUTEXC**, uno per i produttori e uno per i consumatori, entrambi inizializzati a 1

I soliti due semafori, **SPAZIO_DISP** e **MSG_DISP**, usati per la sincronizzazione:

- **SPAZIO_DISP** inizializzato a **DIM** (o al numero di produttori, se minore o uguale a DIM)
- **MSG_DISP** inizializzato a **0**

Prod-Cons con un pool di buffer

```
void Produttore(int* stato, msg* buffer, int sem) {
        int indice = 0;
        Wait_Sem(sem, SPAZIO_DISP); // attende spazio

        Wait_Sem(sem, MUTEXP); // sez. critica
        // trova il primo elemento VUOTO in "indice"
        while (indice<DIM && stato[indice]!=VUOTO)
                indice++;
        stato[indice]=IN_USO;
        Signal_Sem(sem, MUTEXP); // sez. critica

        // Produzione di valore_prodotto . . .
        buffer[indice]=valore_prodotto;
        stato[indice]=PIENO;

        Signal_Sem(sem, MSG_DISP); // segnala i cons
}
```

Prod-Cons con un pool di buffer

```
void Consumatore(int* stato, msg* buffer, int sem) {
        int indice = 0;
        Wait_Sem(sem, MSG_DISP); // attende messaggio

        Wait_Sem(sem, MUTEXC); // sez. critica
        // determina l'indice del primo elemento PIENO
        while (indice<DIM && stato[indice]!=PIENO)
                indice++;
        stato[indice]=IN_USO;
        Signal_Sem(sem, MUTEXC); // sez. critica

        // consumo del messaggio
        msg valore_consumato = buffer[indice];
        stato[indice]=VUOTO;

        Signal_Sem(sem, SPAZIO_DISP); // segnala i prod
}
```

Prod-Cons con un pool di buffer

- Con questa soluzione, i produttori (o i consumatori) accedono alla sezione critica gestita da **MUTEXP** (o **MUTEXC**) solo per acquisire una cella tramite ricerca nel vettore di stato
- Una volta acquisita la cella, possono procedere concorrentemente nella produzione (o consumo)
- I produttori "veloci" potranno terminare prima e segnalare prima i consumatori (e viceversa)

Problema dei Lettori/Scrittori

Due categorie di processi:

- **Lettori**, che leggono un messaggio su di una risorsa condivisa
- **Scrittori**, che scrivono il messaggio dalla risorsa condivisa

Vincoli:

1. i processi **lettori** possono accedere **contemporaneamente** alla risorsa
2. i processi **scrittori** hanno accesso **esclusivo** alla risorsa
3. i **lettori** e **scrittori** si **escludono mutuamente** dall'uso della risorsa

Lettori/Scrittori con starvation degli scrittori

- Le operazioni di lettura sono "protette" dalle procedure di **Inizio_Lettura()** e **Fine_Lettura()**,
- Le operazioni di scrittura sono "protette" da **Inizio_Scrittura()** e **Fine_Scrittura()**.
- Un processo **lettore attende** solo se la risorsa è **occupata** da un processo scrittore, e un processo **scrittore** può accedere alla risorsa solo se questa è **libera**.
- Questa particolare strategia di sincronizzazione può tuttavia provocare condizioni di **attesa indefinita (STARVATION)** per i processi scrittori.

Lettori/Scrittori con starvation degli scrittori

La scrittura "sovrascrive" il dato con un nuovo valore

La lettura non "distrugge" il dato

scrittura

lettura

lettura

scrittura

lettura

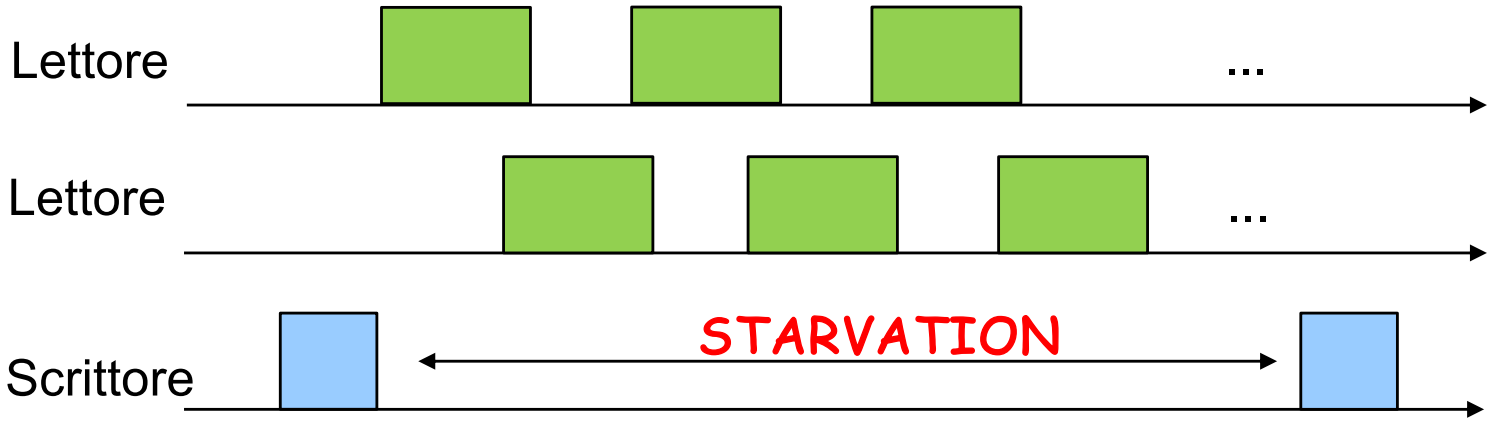
...ATTENDE...
...SI RIATTIVA...

Uno scrittore si sospende se vi è un qualunque altro processo attivo (lettore o altro scrittore)

Un lettore si sospende se vi è uno scrittore (ma non si sospende se ci sono altri lettori)

Lettori/Scrittori con starvation degli scrittori

- La presenza di un lettore permette solo ad altri lettori di entrare, ma non permette l'ingresso agli scrittori
- Finché nel sistema vi è almeno un lettore attivo, lo scrittore subisce la starvation



Lettori/Scrittori con starvation degli scrittori

- Una variable condivisa **NUM_LETTORI** viene usata per contare il numero di lettori che contemporaneamente accedono alla risorsa
- Solo quando NUM_LETTORI = 0, gli scrittori possono accedere (uno alla volta) alla risorsa
- Si utilizzano **2 semafori**:
  - MUTEXL per gestire l'accesso alla variabile NUM_LETTORI in mutua esclusione, da parte dei lettori (inizializzato a 1)
  - SYNCH per garantire la mutua esclusione tra i processi lettori e scrittori e tra i soli processi scrittori (inizializzato a 1)

Lettori/Scrittori con starvation degli scrittori

```
void Processo_Lettore(int sem) {

        Inizio_Lettura(sem);

        // la lettura
        …

        Fine_Lettura(sem);
}
```

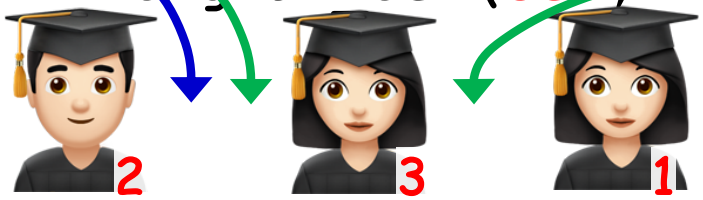
Divideremo l'algoritmo in due parti, in due funzioni diverse
Ogni lettore le chiama subito prima e subito dopo l'operazione di lettura

Lettori/Scrittori con starvation degli scrittori

```
void Inizio_Lettura(int sem) {
        Wait_Sem(sem, MUTEXL);
        Num_Lettori++;
        if (Num_Lettori==1) Wait_Sem (sem, SYNCH);
        Signal_Sem(sem, MUTEXL);
}


void Fine_Lettura(int sem) {
        Wait_Sem(sem, MUTEXL);
        Num_Lettori--;
        if (Num_Lettori==0) Signal_Sem (sem, SYNCH);
        Signal_Sem(sem, MUTEXL);
}
```

Lettori/Scrittori con starvation degli scrittori

```
void Inizio_Lettura(int sem) {
        Wait_Sem(sem, MUTEXL);
        Num_Lettori++;
        if (Num_Lettori==1) Wait_Sem (sem, SYNCH);
        Signal_Sem(sem, MUTEXL);
}
```



Il primo lettore "occupa" il semaforo SYNCH

Supponendo che non vi siano scrittori attivi…
…tutti i lettori eseguono senza bloccarsi

Lettori/Scrittori con starvation degli scrittori

In attesa su MUTEXL

```
void Inizio_Lettura(int sem)
        Wait_Sem(sem, MUTEXL);
        Num_Lettori++;
        if (Num_Lettori==1) Wait_Sem (sem, SYNCH);
        Signal_Sem(sem, MUTEXL);
}
```

Pone Num_Lettori a 1, si mette in attesa su SYNCH

Supponiamo che il buffer sia già occupato da un processo scrittore...

Lettori/Scrittori con starvation degli scrittori

```
void Inizio_Lettura(int sem) {
        Wait_Sem(sem, MUTEXL);
        Num_Lettori++;
        if (Num_Lettori==1) Wait_Sem (sem, SYNCH);
        Signal_Sem(sem, MUTEXL);
}
```

Anche gli altri si riattivano e leggono…

Si riattiva, libera la sezione critica, effettua la lettura…

```
void Fine_Lettura(int sem) {
        Wait_Sem(sem, MUTEXL);
        Num_Lettori--;
        if (Num_Lettori==0) Signal_Sem (sem, SYNCH);
        Signal_Sem(sem, MUTEXL);
}
```

Lettori/Scrittori con starvation degli scrittori

```
void Inizio_Lettura(int sem) {
      Wait_Sem(sem, MUTEXL);
      Num_Lettori++;
      if (Num_Lettori==1) Wait_Sem (sem, SYNCH);
      Signal_Sem(sem, MUTEXL);
}


void Fine_Lettura(int sem) {
      Wait_Sem(sem, MUTEXL);
      Num_Lettori--;
      if (Num_Lettori==0) Signal_Sem (sem, SYNCH);
      Signal_Sem(sem, MUTEXL);
}
```

L'ultimo lettore riattiva un (eventuale) scrittore in attesa

Lettori/Scrittori con starvation degli scrittori

```
void Inizio_Scrittura(int sem) {
      Wait_Sem(sem, SYNCH);
}

void Fine_Scrittura(int sem) {
      Signal_Sem(sem, SYNCH);
}
```

Starvation degli scrittori: mentre uno scrittore è sospeso sul semaforo SYNCH, altri lettori possono accedere alla risorsa, ritardando indefinitamente lo scrittore

Lettori/Scrittori con starvation di entrambi

- Anche per gli scrittori si può ottenere un comportamento analogo, introducendo una variabile **NUM_SCRITTORI**
- In questo caso occorrono 4 semafori, tutti inizializzati a 1:
  - **MUTEXL** per gestire l'accesso alla variabile NUM_LETTORI in mutua esclusione, da parte dei lettori
  - **MUTEXS** per gestire l'accesso alla variabile NUM_SCRITTORI in mutua esclusione, da parte degli scrittori
  - **MUTEX** per gestire l'accesso in mutua esclusione alla risorsa condivisa da parte degli scrittori
  - **SYNCH** per garantire la mutua esclusione tra i processi lettori e scrittori

Lettori/Scrittori con starvation di entrambi

```
void Inizio_Scrittura(int sem) {
        Wait_Sem(sem, MUTEXS);
        Num_Scrittori++;
        if (Num_Scrittori==1) Wait_Sem(sem, SYNCH);
        Signal_Sem(sem, MUTEXS);

        Wait_Sem(sem, MUTEX);
}

void Fine_Scrittura(int sem) {

        Signal_Sem(sem, MUTEX);

        Wait_Sem(sem, MUTEXS);
        Num_Scrittori--;
        if (Num_Scrittori==0) Signal_Sem (sem, SYNCH);
        Signal_Sem(sem, MUTEXS);
}
```

Lettori/Scrittori con starvation di entrambi

```
void Inizio_Scrittura(int sem) {
      Wait_Sem(sem, MUTEXS);
      Num_Scrittori++;
      if (Num_Scrittori==1) Wait_Sem(sem, SYNCH);
      Signal_Sem(sem, MUTEXS);
      Wait_Sem(sem, MUTEX);
}

void Fine_Scrittura(int sem) {
      Signal_Sem(sem, MUTEX);
      Wait_Sem(sem, MUTEXS);
      Num_Scrittori--;
      if (Num_Scrittori==0) Signal_Sem (sem, SYNCH);
      Signal_Sem(sem, MUTEXS);
}
```

È uguale all'algoritmo per i lettori, ma con la **aggiunta di un ulteriore sezione critica** (gli scrittori devono escludersi tra di loro)