Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2016-17

Pietro Frasca

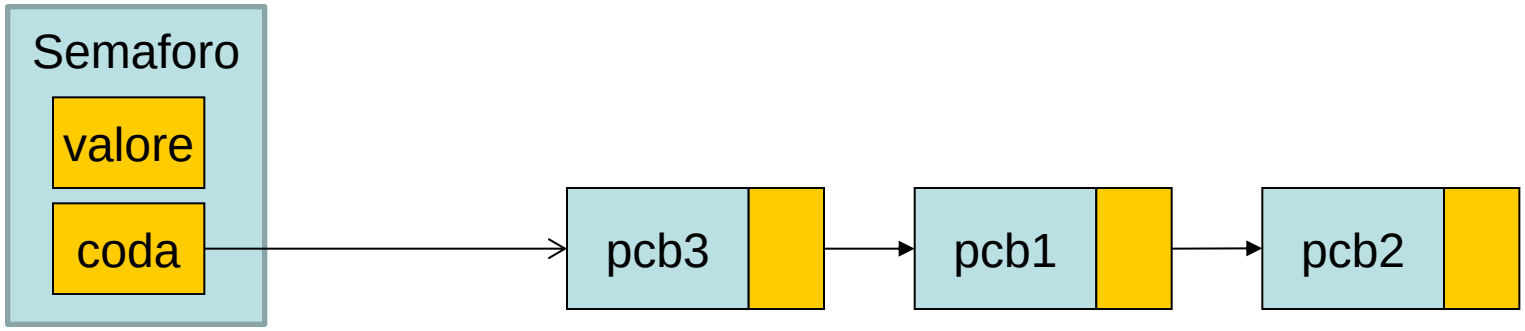
Lezione 9

Giovedì 10-11-2016

Semafori

- un semaforo **s** è una struttura dati gestita principalmente dalle chiamate di sistema **wait(s)** e **signal(s) e** dalla chiamata di inizializzazione**.**
- 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 **wait è bloccante** se **s.valore=0**.
- Le chiamate wait e signal devono essere 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:

```
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
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 chiamate 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 mutex, è necessario che i processi utilizzino le funzioni lock e unlock, secondo il protocollo seguente:

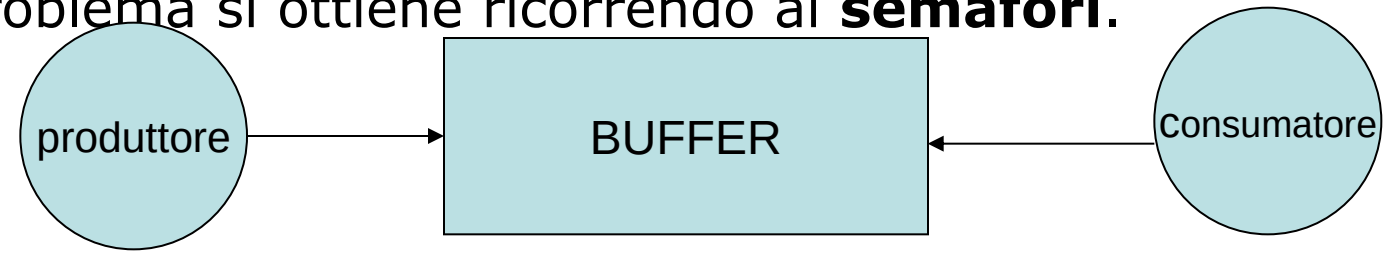
```
lock(x);
    wait(mutex);
unlock(x);
    <sezione critica>;
lock(x);
    signal(mutex);
unlock(x);
```

- 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 tra processi

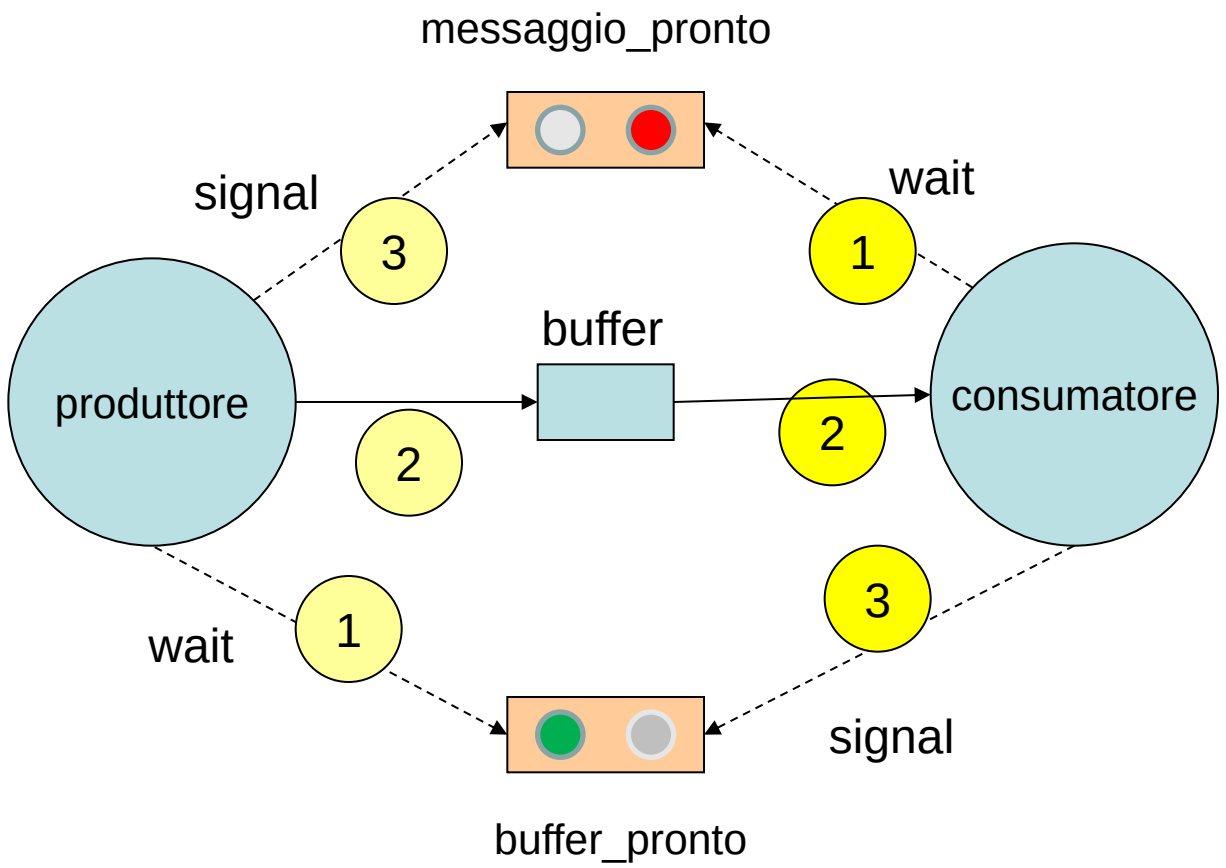
- Il paradigma del **produttore-consumatore** è spesso usato per la comunicazione tra processi.
- In tale modello, un processo detto **produttore** genera un messaggio e lo scrive in un area di memoria (buffer) che contiene un solo messaggio alla volta. Un processo, detto **consumatore** preleva dal 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 **buffer_pronto** e **messaggio_pronto** con le condizioni iniziali:
  - **buffer_pronto.valore=1** (inizialmente il buffer è vuoto)
  - **messaggio_pronto.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

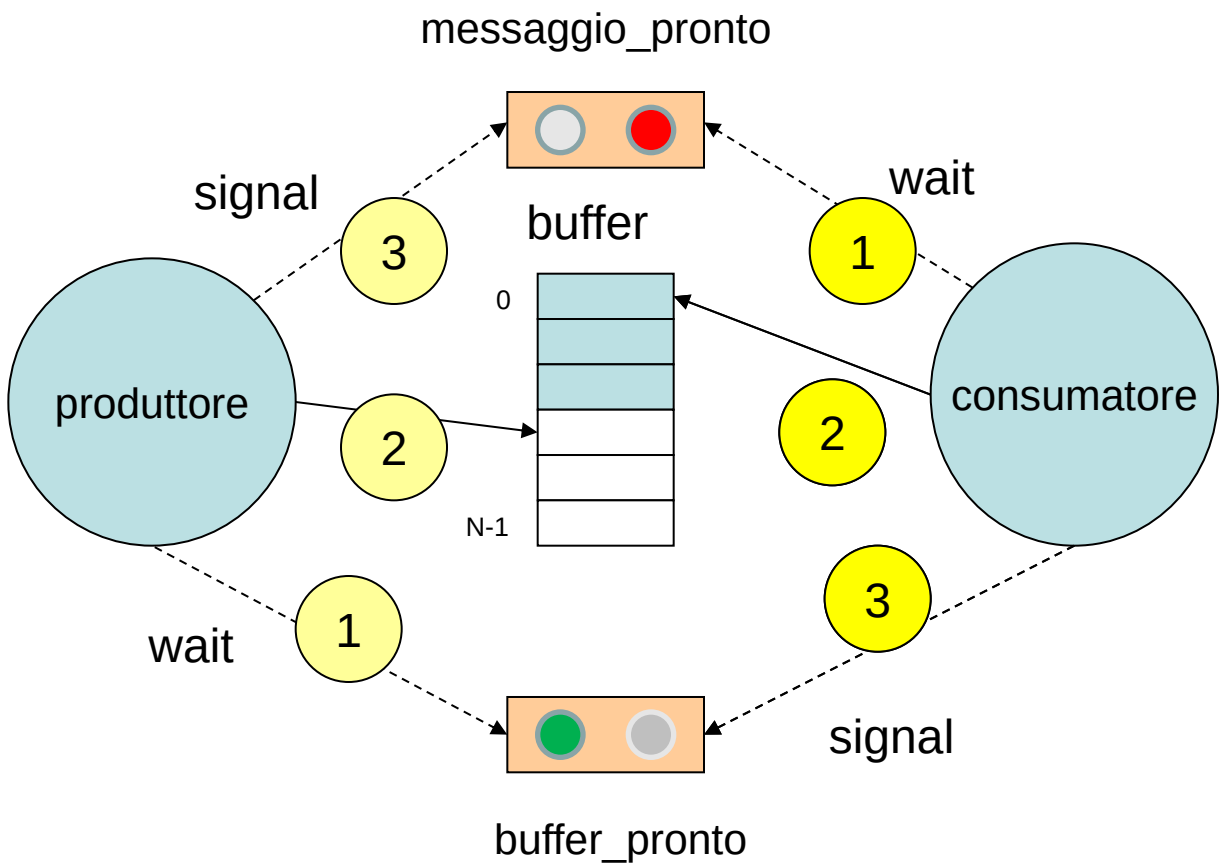
```
void produttore (){
  do {
     <produzione nuovo messaggio>
     wait (buffer_pronto);
     <inserimento del messaggio nel buffer>
     signal(messaggio_pronto);
  } while (!fine);
}

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

Soluzione al problema della comunicazione

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 **buffer_pronto** e **messaggio_pronto** con le condizioni iniziali:
  buffer_pronto.valore=N;
  messaggio_pronto.valore=0;.



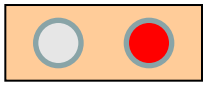
Sezione critica 2

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

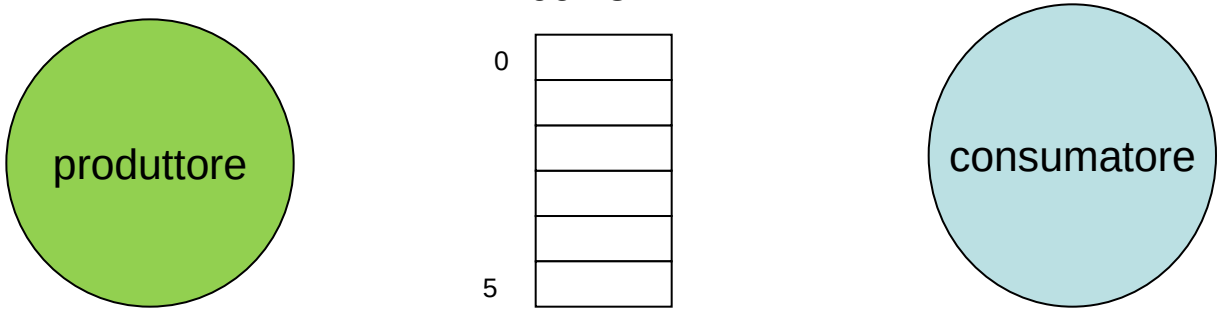
Sezione critica 2

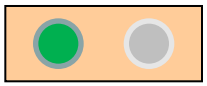
```
x=buffer[leggi];
leggi=(leggi+1)%N
```

messaggio_pronto.valore=0



buffer





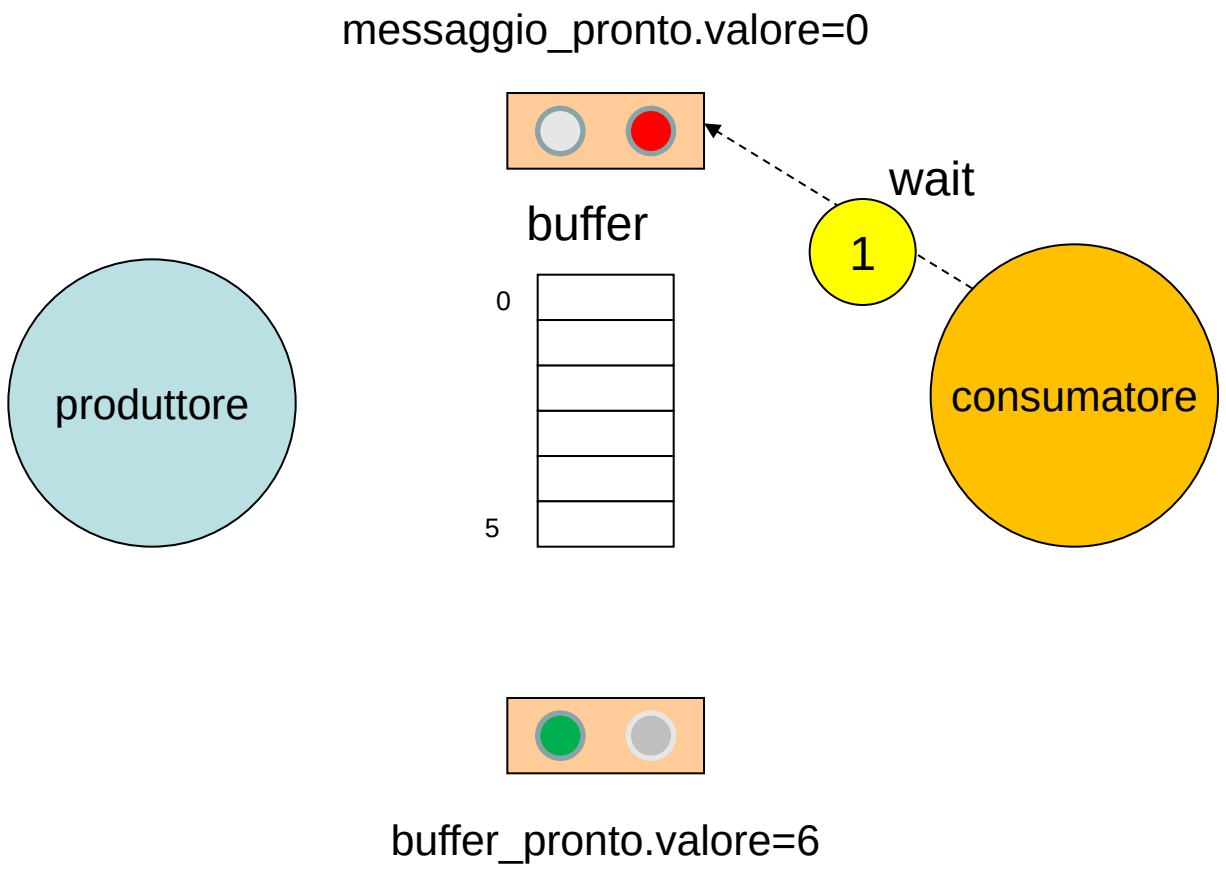
buffer_pronto.valore=6

Sezione critica

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

Sezione critica

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

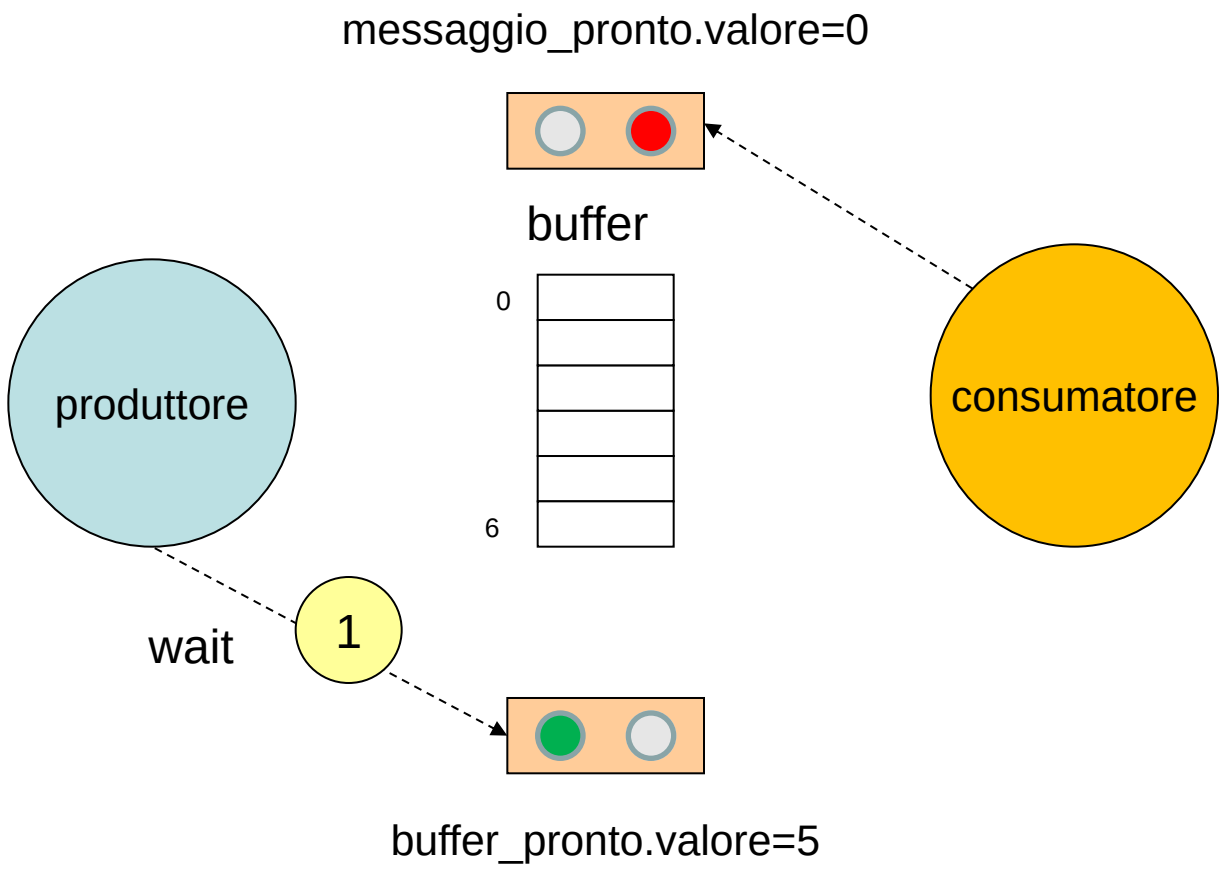


Sezione critica

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

Sezione critica

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



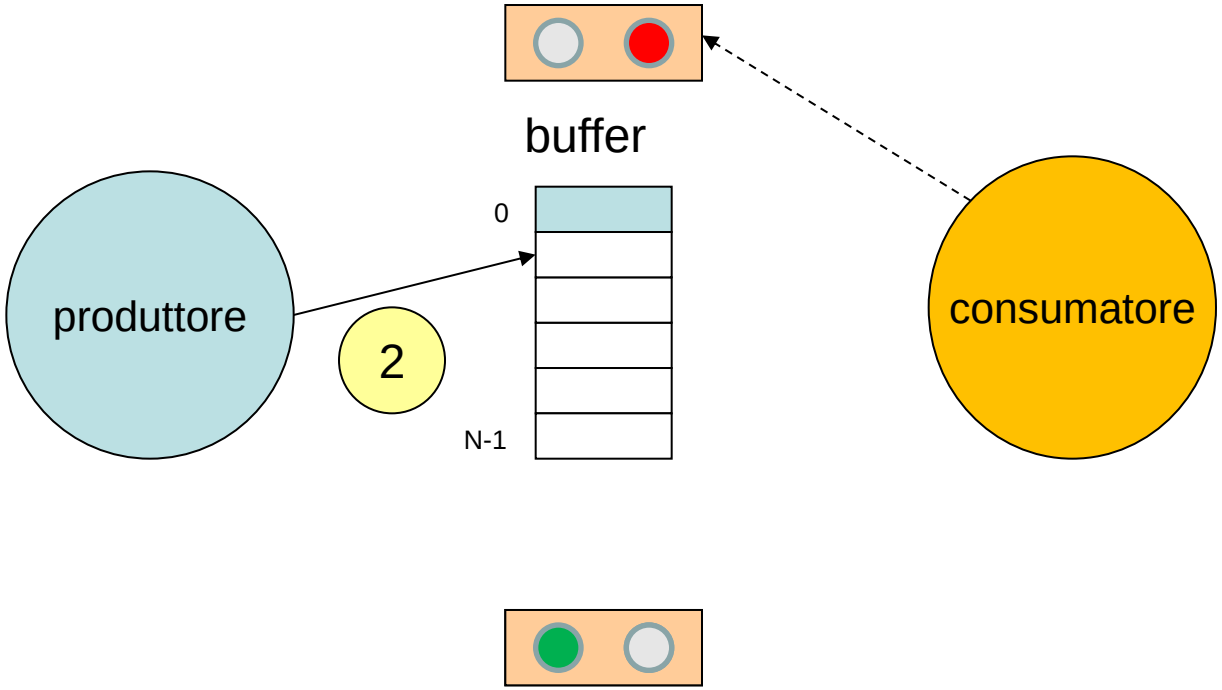
Sezione critica

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

Sezione critica

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

messaggio_pronto.valore=0



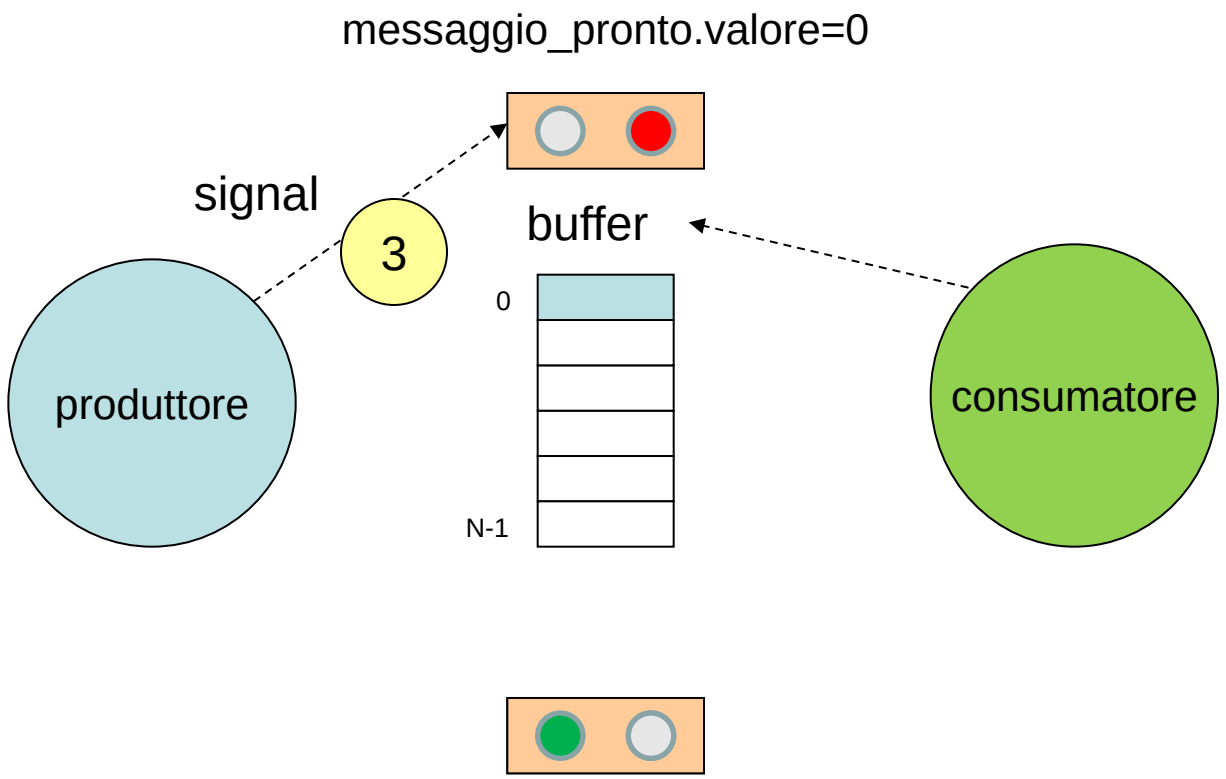
buffer_pronto.valore=5

Sezione critica 2

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

Sezione critica

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

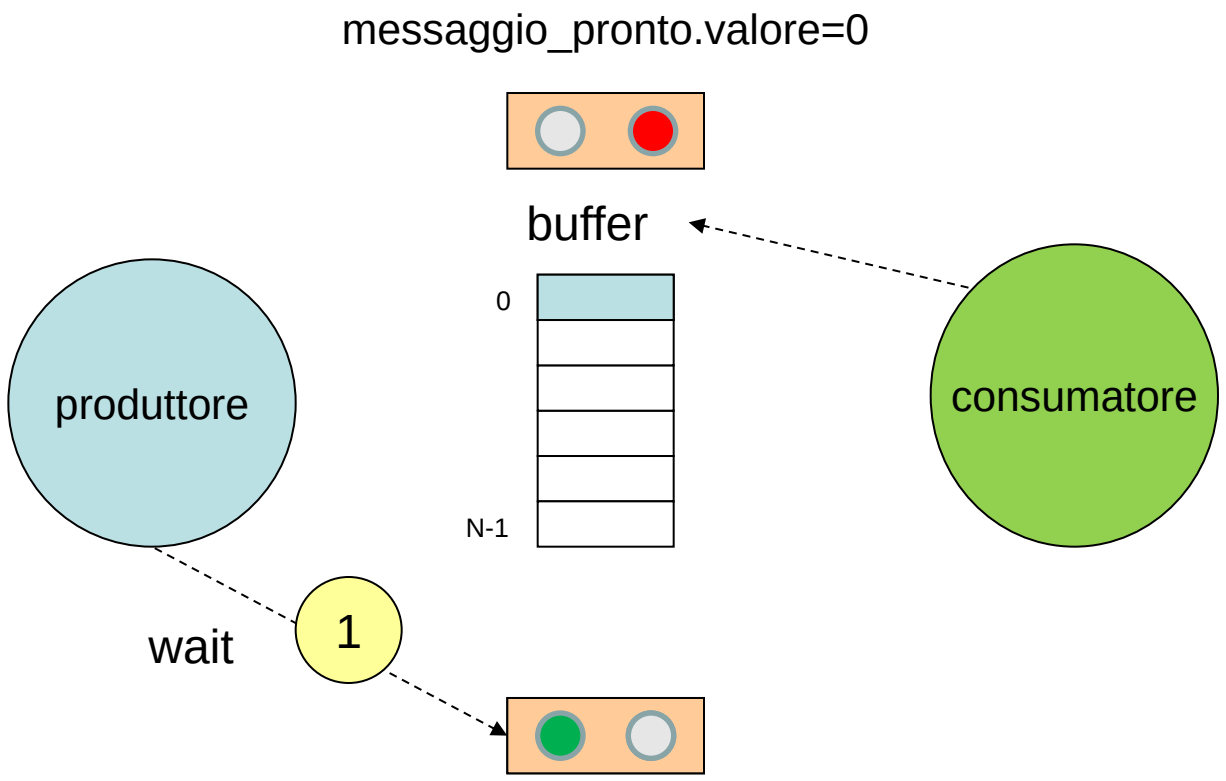


Sezione critica

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

Sezione critica

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



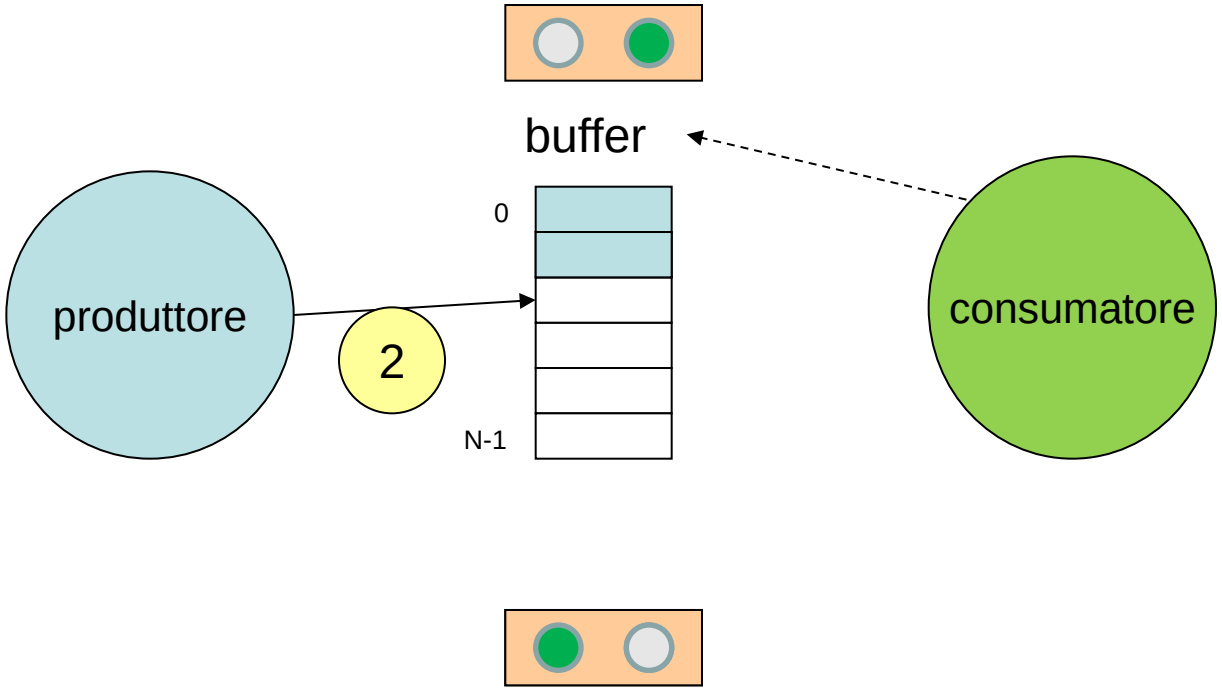
Sezione critica

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

Sezione critica

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

messaggio_pronto.valore=0



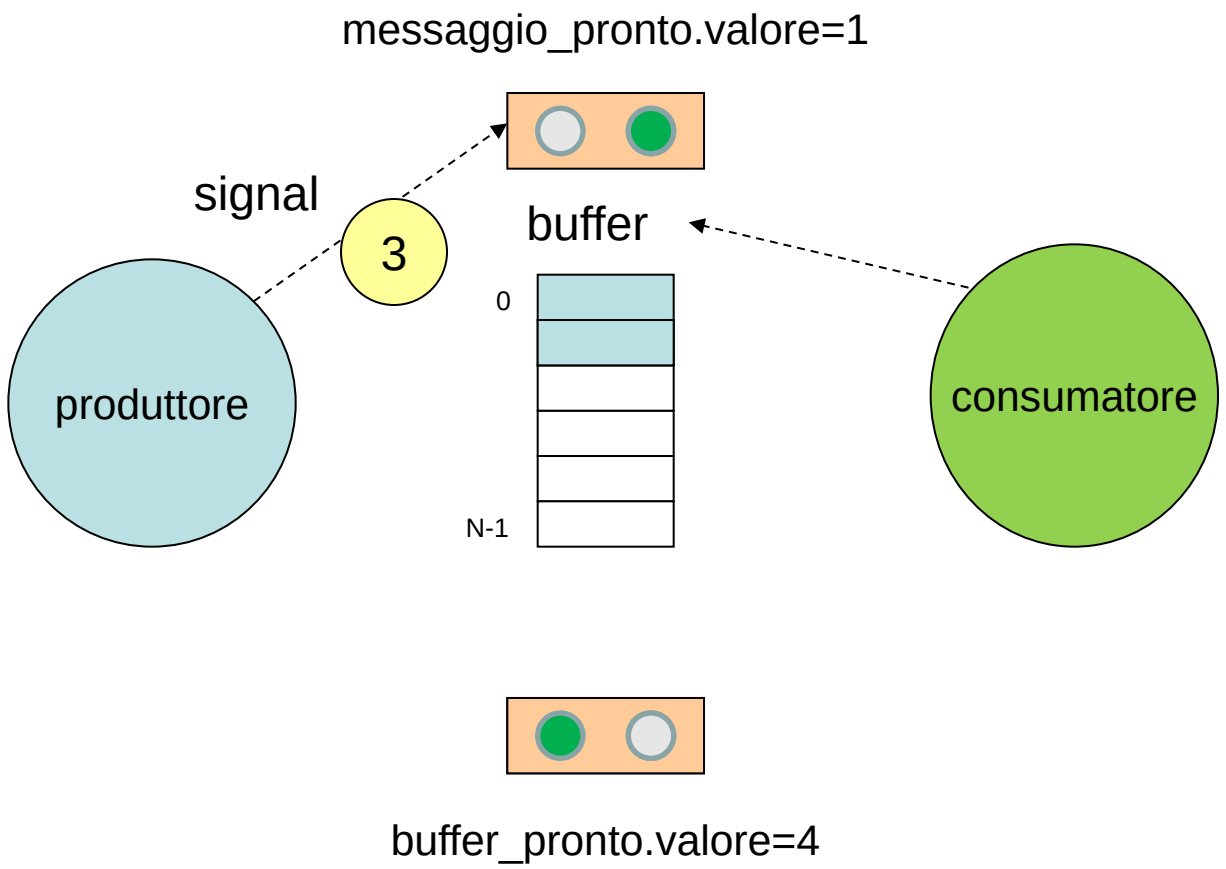
buffer_pronto.valore=4

Sezione critica 2

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

Sezione critica

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

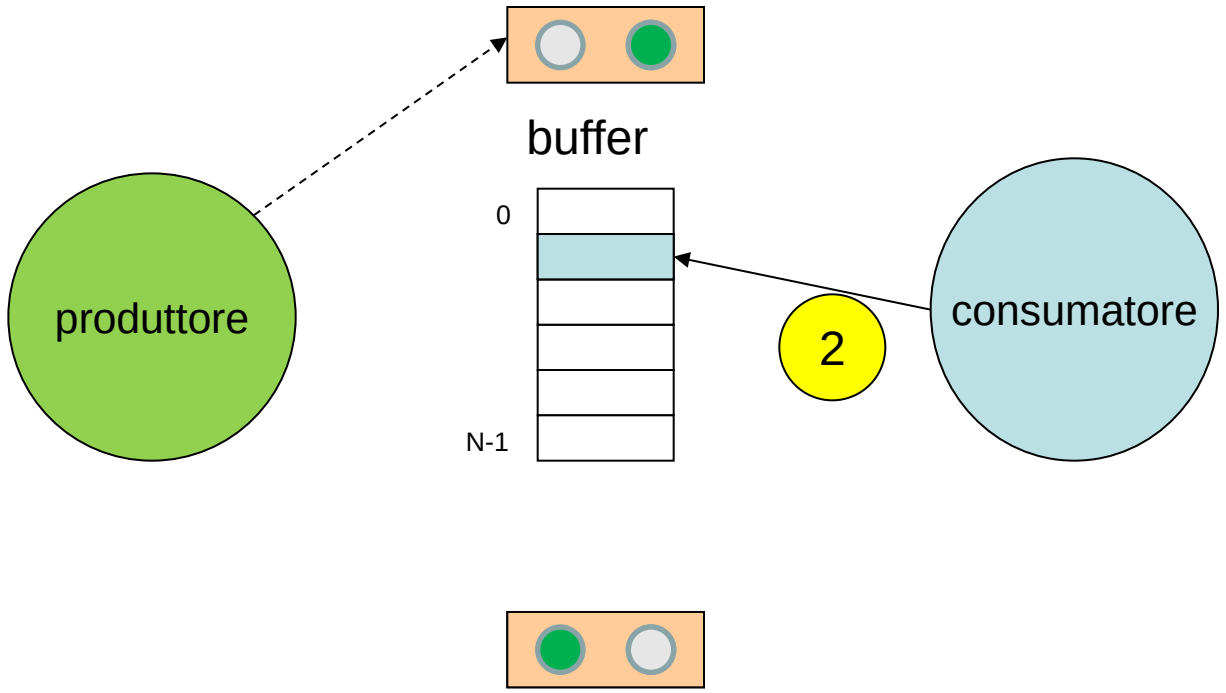


Sezione critica

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

Sezione critica

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

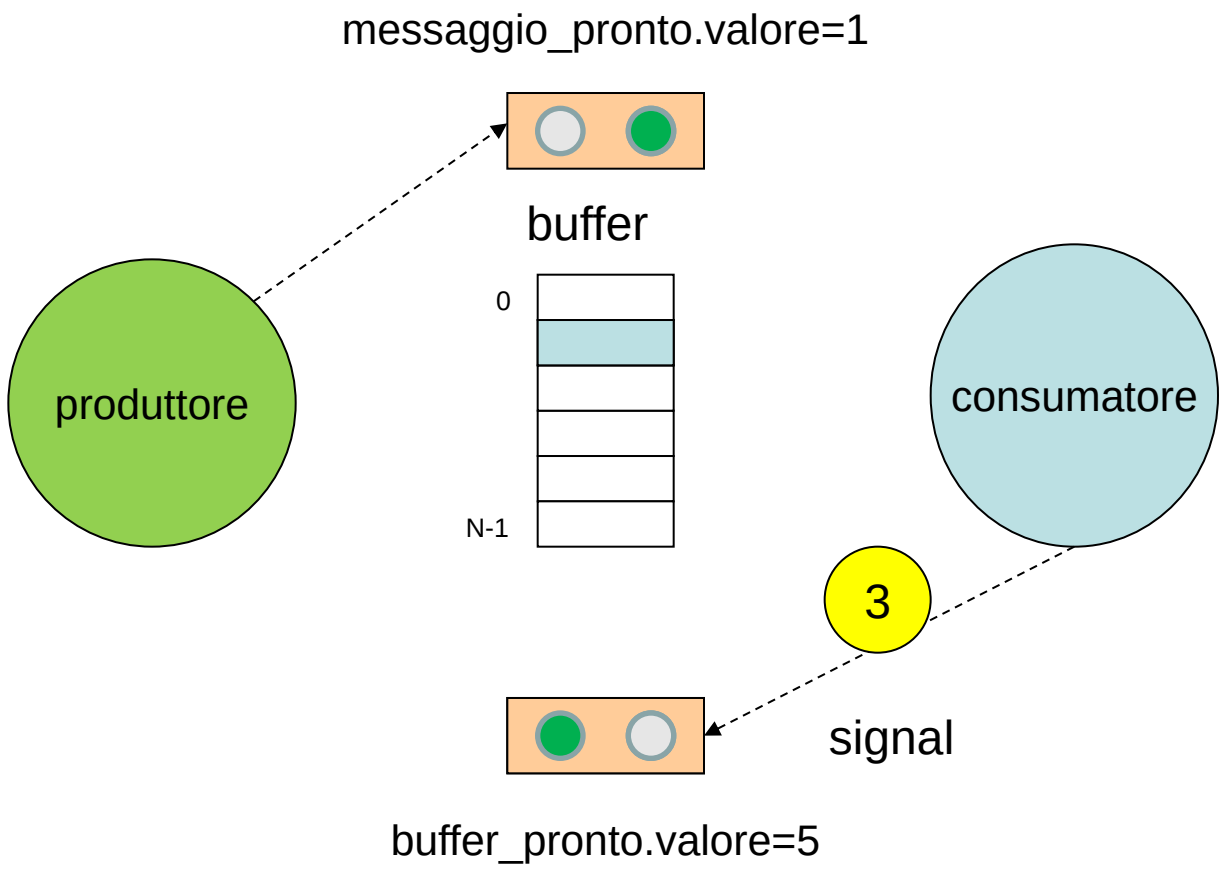


Sezione critica

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

Sezione critica 2

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

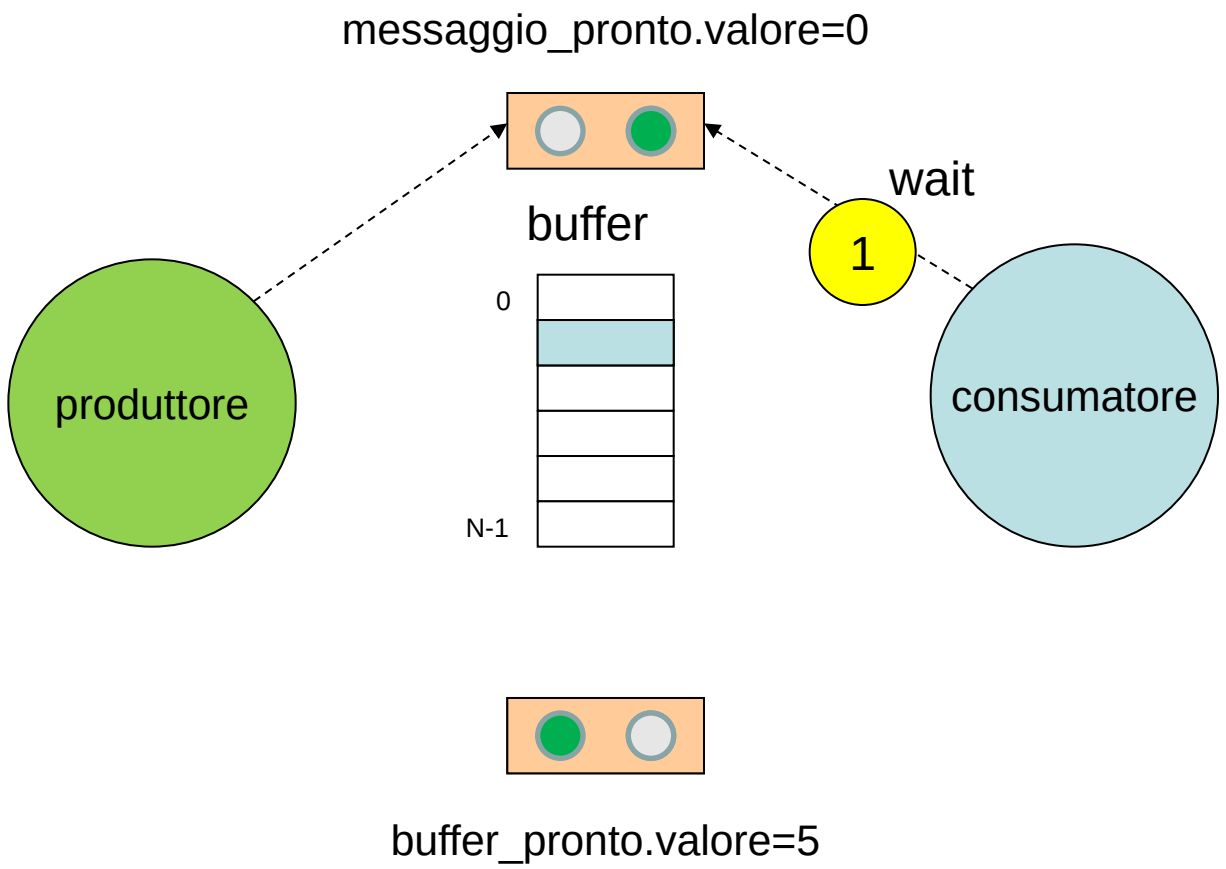


Sezione critica

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

Sezione critica

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

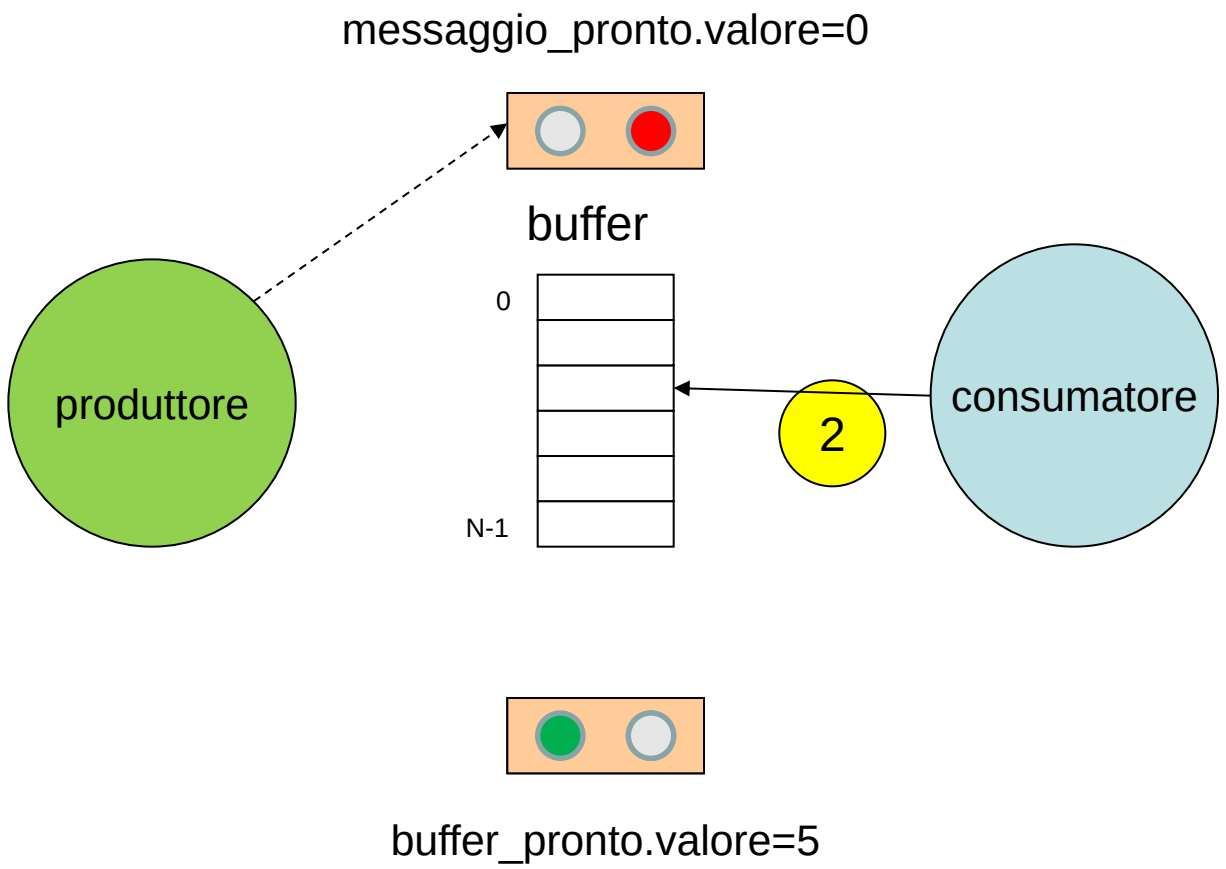


Sezione critica

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

Sezione critica

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

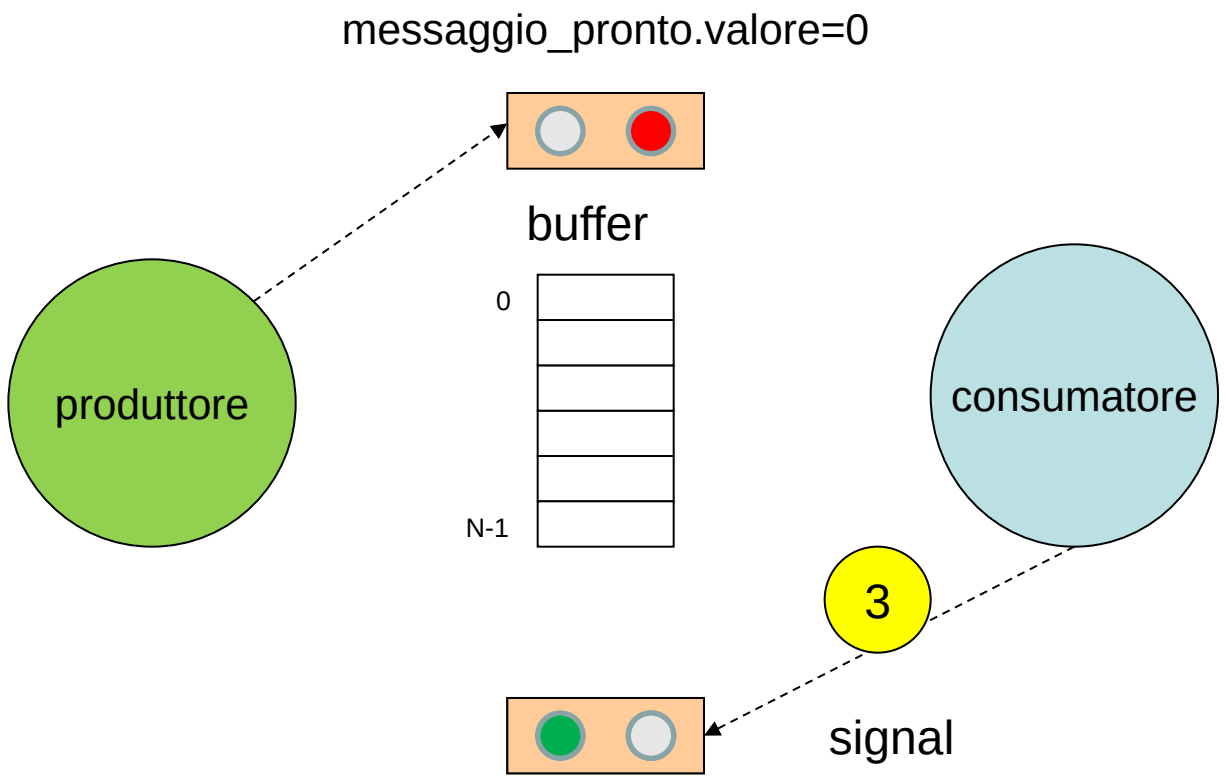


Sezione critica

```
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

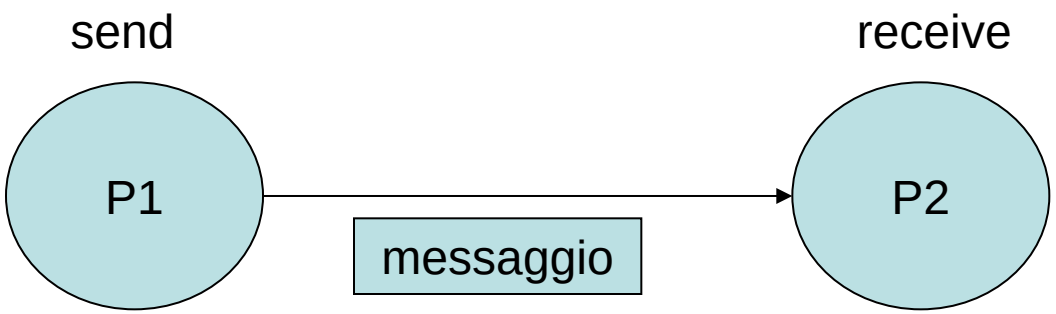
```
x=buffer[leggi];
leggi=(leggi+1)%N
```

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

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

Sincronizzazione dei processi con scambio di messaggi (message passing)

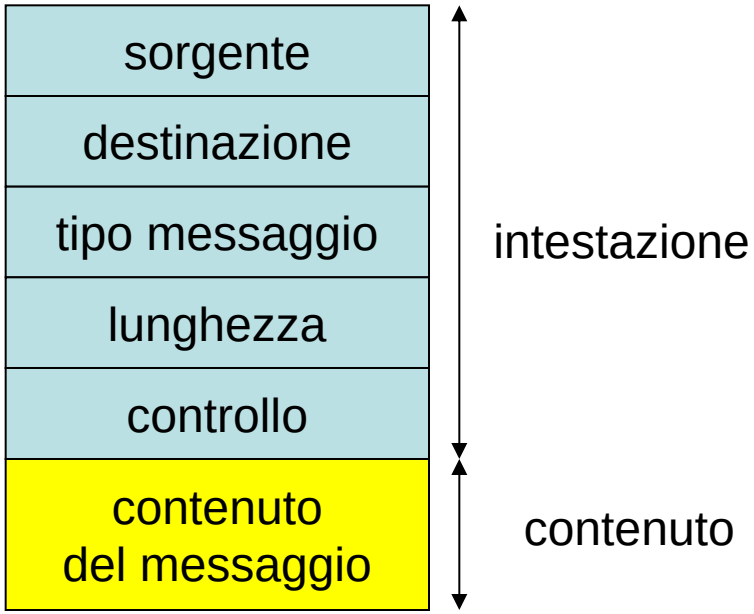
- Abbiamo visto come, in un ambiente a memoria comune, l'uso dei semafori consente di risolvere problemi di cooperazione e competizione tra processi.
- In un ambiente a memoria locale l'interazione tra processi avviene mediante scambio di messaggi tra i processi stessi.



- Lo scambio di messaggi è molto diffuso in ambienti distribuiti dove i processi comunicanti sono eseguiti in calcolatori diversi connessi tra loro mediante una rete di comunicazione.
- Tuttavia il *message passing* è usato anche in sistemi a memoria condivisa, sia a singolo processore che multiprocessore.
- In quest'ultimo caso il canale di comunicazione è costituto da un'area di memoria, gestita dal sistema operativo, nella quale vengono scritti e letti i messaggi.
- I processi possono comunicare tra loro mediante un insieme di chiamate di sistema che viene indicato con il termine **IPC** *(**Inter-Process-Communication**)*.
- Due basilari chiamate di sistema di IPC sono **`send`** e **`receive`**:

```
send(destinatario, messaggio);
receive(mittente, messaggio);
```

- La **send** invia un messaggio ad un processo destinatario e la **receive** riceve un messaggio da un processo mittente.
- Un messaggio ha una struttura ben precisa. Tipicamente è composto da due parti, l'***intestazione*** ed il **contenuto (payload)**. Campi tipici dell'intestazione sono l'identificatore del mittente e del destinatario, il tipo di messaggio, la lunghezza del messaggio, ecc. Il contenuto include il messaggio vero e proprio.



Soluzione al problema della comunicazione tra processi

- Con la comunicazione a scambio di messaggi i processi possono comunicare tra loro ***direttamente*** o ***indirettamente***.

Comunicazione diretta

- Si ha una **comunicazione diretta** tra processi quando è necessario indicare esplicitamente nelle chiamate **send** e **receive** i nomi dei processi mittente e destinatario. Se ad esempio il processo P1 invia un messaggio a P2 i due processi eseguiranno rispettivamente:

```
send (P2,messaggio); // eseguita da P1
receive (P1,messaggio); // eseguita da P2
```

- La comunicazione diretta, si dice ***simmetrica*** quando si realizza mediante un canale che è associato in modo univoco ai due processi.

La figura mostra uno schema di due processi produttore e consumatore nel caso di comunicazione diretta simmetrica.

PRODUTTORE

```
pid cons = …
main(){
  Messaggio mes;
  do {
    produci (mes);
    send (cons,mes);
  while (!fine)
}
```

CONSUMATORE

```
pid prod =…
main(){
  Messaggio mes;
  do {
    receive (prod,mes);
    consuma (mes)
  while (!fine)
}
```

Schema di comunicazione diretta simmetrica

- La comunicazione **diretta asimmetrica** si ha quando più processi mittente inviano un messaggio ad un unico processo destinatario.
- In questo caso, ciascun processo mittente specifica esplicitamente il destinatario, mentre il processo ricevente non può specificare il mittente in quanto più processi possono inviargli messaggi.
- Un esempio di comunicazione diretta asimmetrica si ha quando un processo, utilizzato per la gestione di una risorsa, come ad esempio un disco o un dispositivo, riceve richieste da più processi.
- In questo caso le chiamate hanno la forma:

  send (PS,messaggio); // eseguita dai Pi (client)
  receive (id,messaggio); // eseguita da PS
  (processo server)

- la **send** è eseguita da uno dei processi **Pi** che invia un messaggio a **PS** (ad esempio, per richiedergli l'esecuzione di un'operazione) e la **receive** è eseguita da PS con **id** che, di volta in volta, assume il nome del processo con cui è avvenuta la comunicazione.

- L' **id** viene estratto dall'intestazione del messaggio e può essere utilizzato per rispondere al mittente del messaggio stesso.
- La figura mostra uno schema di due processi produttore e consumatore nel caso di comunicazione diretta asimmetrica.

PRODUTTORE

```
pid cons = …
main(){
  Messaggio mes;
  do {
    produci (mes);
    send (cons,mes);
  while (!fine)
}
```

CONSUMATORE

```
main(){
  Messaggio mes;
  pid id;
  do {
    receive (id,mes);
    consuma (mes)
  while (!fine)
}
```

Schema di comunicazione diretta asimmetrica

• La figura mostra uno schema di due processi produttore e consumatore nel caso di comunicazione diretta asimmetrica in cui il consumatore invia un messaggio di risposta al produttore.

PRODUTTORE

```
pid cons = …
main(){
  Messaggio mes, mes_ris;
  do {
    produci (mes);
    send (cons,mes);
    receive(cons, mes_ris);
  while (!fine)
}
```

CONSUMATORE

```
main(){
  Messaggio mes, mes_ris;
  pid id;
  do {
    receive (id,mes);
    consuma (mes);
    send(id, mes_ris);
  while (!fine)
}
```

Schema di comunicazione diretta asimmetrica

- L'identificatore **pid** *(**process identifier**)* rappresenta un tipo di variabile utilizzato per identificare un processo.
- Nei sistemi a memoria comune può coincidere con il **pid** del processo assegnato al momento della sua creazione.
- Nelle reti, ad esempio Internet, un processo è identificato dal numero IP del computer dove esso è in esecuzione e da un numero di porta.

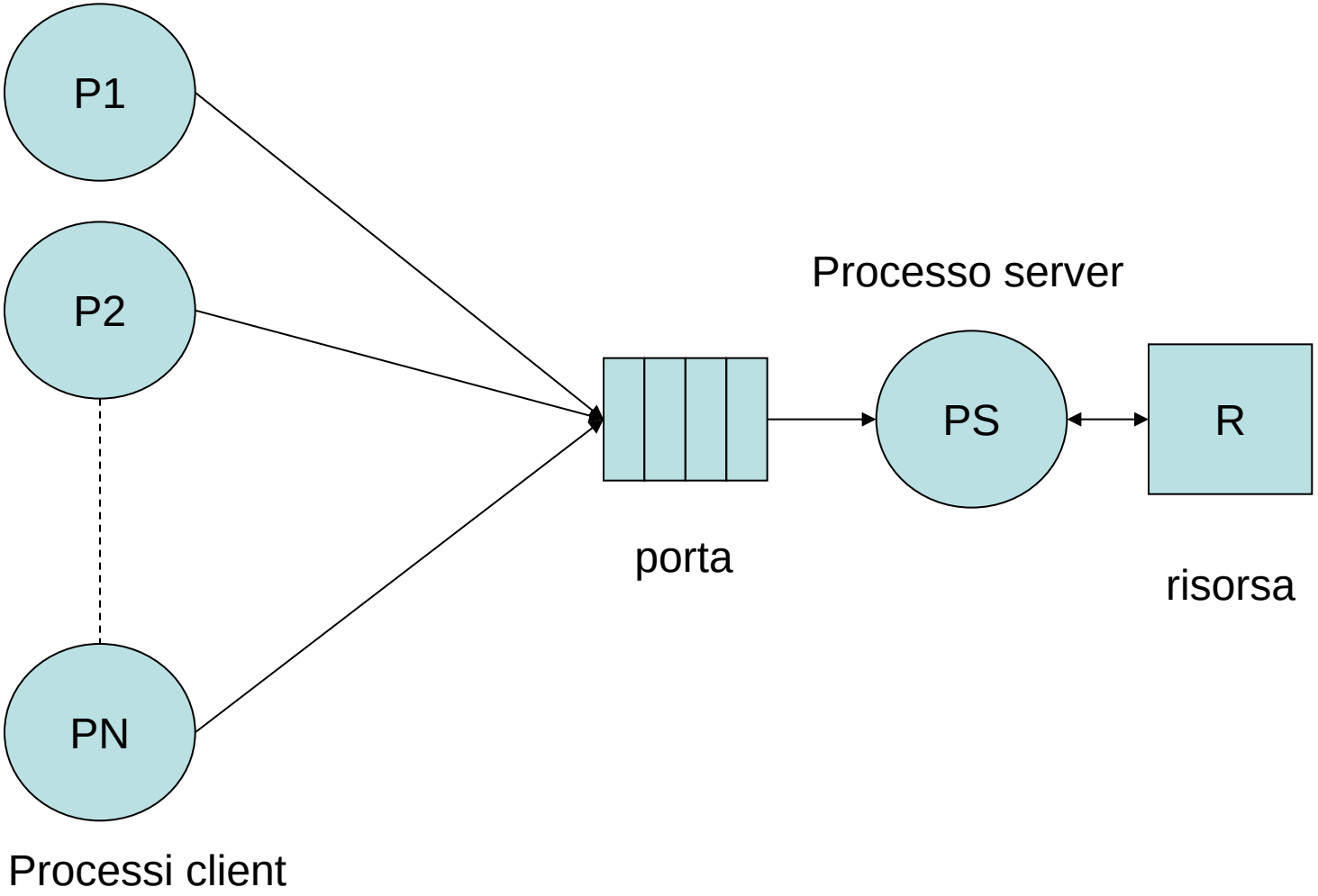
Comunicazione indiretta

- La ***comunicazione indiretta*** si ha quando i processi si scambiano messaggi mediante una struttura dati detta ***porta***.
- Nel caso di sistemi mono o multiprocessore con modello a memoria locale, la porta, che prende il nome di **mailbox**, è creata e gestita dal sistema operativo. La comunicazione tra due processi P1 e P2 avviene nel modo seguente:

```
send (mailbox, messaggio) ; // eseguita da P1
receive (mailbox, messaggio); // eseguita da P2
```

- La chiamata send è utilizzata da P1 per inviare un messaggio alla mailbox e la receive è utilizzata da P2 per leggere un messaggio dalla mailbox.
- Nell'ipotesi che il modello a memoria locale sia realizzato su sistemi distribuiti, le porte sono create e gestite dai sistemi operativi sui processori su cui sono attivi i processi.
- Le possibili relazioni tra mittenti e riceventi possono essere:
  - ***uno-a-uno***,
  - ***uno-a-molti***,
  - ***molti-a-uno***
  - ***molti-a-molti***
- In particolare, la relazione ***moltia-uno*** *è tipica* di collegamenti ***client-server***.

- Il modello client-server prevede che un processo server si comporti come gestore di risorse.
- I client per utilizzare una risorsa inviano un messaggio di richiesta di servizio alla porta associata al processo server.
- Il processo server analizza le varie richieste di servizio contenute nella porta, ne sceglie una in base a determinate politiche di gestione, ed esegue operazioni sulla risorsa R rispondendo, se previsto, al relativo processo client.



Modello client-server

Sincronizzazione tra processi comunicanti

- La sincronizzazione tra processi mediante l'uso di send e receive, presenta varie alternative.
- La **send**, può funzionare in una delle due modalità:

  send asincrona (non bloccante)
  send sincrona (loccante)

- La receive può funzionare nelle due modalità:

  receive sincrona (bloccante)
  receive asincrona (non bloccante)

- La **send asincrona** consente al processo di continuare la sua esecuzione subito dopo l'esecuzione della send stessa.
- La **send sincrona** blocca il processo mittente fino al ricevimento di un messaggio da parte del processo destinatario.
- La **receive bloccante** causa la sospensione del processo che la esegue se non ci sono messaggi in arrivo; all'arrivo di un messaggio il processo viene risvegliato e continua la sua esecuzione.
- La **receive non bloccante** consente di continuare l’esecuzione del processo anche in assenza di messaggi.
- Sono possibili diverse combinazioni di send e receive tra quelle illustrate.
- Ad esempio, nel **modello client-server**, sia la send che la receive sono **bloccanti**. Il processo server, infatti, non può eseguire la sua azione in assenza di una specifica richiesta

- La send non bloccante consente al processo che l'ha chiamata di continuare la sua esecuzione. Può risultare valida se il mittente non deve attendere una risposta dal processo ricevente.
- L'uso della receive in modalità non bloccante può essere utile per analizzare, secondo un ordine stabilito, lo stato di più comunicazioni instaurate con vari client. In tal caso se su alcune connessioni non giungono messaggi si ha una poco efficiente forma di attesa attiva.