

# Programmazione concorrente nel Modello ad Ambiente Globale

# Interazione nel modello ad Ambiente Globale (Memoria Comune)

Se la macchina concorrente e' organizzata secondo il modello ad ambiente globale (o modello a memoria comune):

- ❑ **PROCESSO=THREAD**
  - ❑ Comunicazione attraverso dati/risorse (*oggetti*) condivisi
  - ❑ Necessita' di sincronizzare gli accessi agli oggetti condivisi
- 
- Il linguaggio concorrente offre **costrutti** per esprimere la soluzione a problemi di sincronizzazione (es: semafori, regioni critiche, monitor ecc.)
  - Il nucleo realizza **meccanismi** di sincronizzazione

# Processi & risorse

- Ogni applicazione concorrente può essere rappresentata come un insieme di componenti, suddiviso in due sottoinsiemi disgiunti:
  - **processi** (componenti attivi)
  - **risorse** (componenti passivi).

**Risorsa:** Qualunque oggetto, fisico o logico, di cui un processo necessita per portare a termine il suo compito.

- Le risorse sono raggruppate in **classi**; ogni classe identifica l'insieme di *tutte e sole* le operazioni che un processo può eseguire per operare su risorse di quella classe.
- Il termine *risorsa* si identifica con quello di *struttura dati* allocata nella memoria comune.

**Risorsa privata di un processo P** (o **locale** a P): P è il *solo* processo che può eseguire operazioni sulla risorsa.

**Risorsa comune (o globale)**: è una risorsa su cui *più processi* possono operare.

→ Nel modello a memoria comune i processi interagiscono *esclusivamente* operando su **risorse comuni** (competizione e cooperazione).

- **Meccanismo di controllo degli accessi**: è necessario definire quali processi ed in quali istanti possono *correttamente* accedere alla risorsa.

# Gestore di una risorsa

- **Allocatore o gestore di una risorsa R:**  
è l'entità che ha il compito di definire in ogni istante  $t$  l'insieme  $SR(t)$  dei processi che possono accedere ad R in quell'istante.
- In base alla cardinalità di  $SR(t)$ , diremo che R è:
  - Una Risorsa **dedicata**, se  $Sr(t)$  contiene al più un processo
  - Una Risorsa **condivisa**, se esiste  $t$ :  $Sr(t)$  contiene più processi

# Allocazione delle risorse

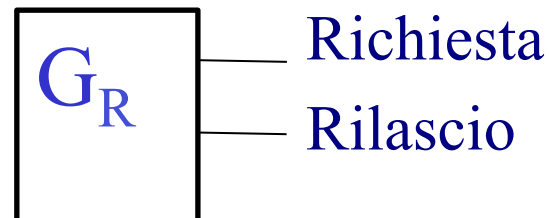
## Risorsa allocata staticamente.

- Il gestore di R definisce l'insieme  $SR$  all'istante  $T_0$  (istante iniziale dell'elaborazione) senza modificarlo durante l'elaborazione  $\rightarrow \forall t, SR(t)=SR(T_0)$
- Il gestore della risorsa è il *programmatore* che in base alle *scope rules* del linguaggio stabilisce quali processi possano vedere e quindi elaborare la risorsa.
- Il compito di controllare gli accessi è svolto dal *compilatore* che, secondo le regole di visibilità, assicura che *soltanto* i processi ai quali la risorsa è stata allocata possano accedervi.

# Allocazione delle Risorse

## Risorsa allocata dinamicamente

- L'insieme  $Sr$  è variabile nel tempo:
  - a) Risorsa **dedicata**.  $Sr(t)$  contiene al più un processo
  - b) Risorsa **condivisa**.  $Sr(t)$  contiene più processi contemporaneamente
- Il gestore della risorsa  $R$  opera a *tempo di esecuzione*. Nel modello a memoria comune il gestore è una *risorsa*  $G_R$  (nel modello a scambio di messaggi è un *processo*):



- $Sr(t)$  è inizialmente vuoto. Per operare sulla risorsa il processo *deve chiedere il permesso al gestore*:  
 **$G_R.Richiesta()$** ;
- Il gestore può accettare, ritardare o rifiutare la richiesta .

# Allocazione delle Risorse

→ *Schema logico seguito da ogni processo che vuole accedere a una risorsa:*

- 1) *richiesta* della risorsa,
- 2) *uso*
- 3) *rilascio* del diritto di accedere.

$G_R$ .Richiesta();  
<uso della Risorsa R>  
 $G_R$ .Rilascio();



	risorsa dedicata	risorsa condivisa
risorsa allocata staticamente	<i>privata</i>	<i>comune</i> ai processi cui la risorsa è allocata
risorsa allocata dinamicamente	<i>comune</i>	<i>comune</i>

Una risorsa allocata staticamente e dedicata ad un solo processo è una *risorsa privata*.

In tutti gli altri casi le risorse sono *comuni*, o perché condivise tra più processi o perché dedicate, ma *dinamicamente*, a processi diversi in tempi diversi.

# Il problema del deadlock

Quando due o più processi competono per l'uso di risorse comuni, è possibile incorrere in situazioni di stallo (**deadlock**, o blocco critico)

## Definizione

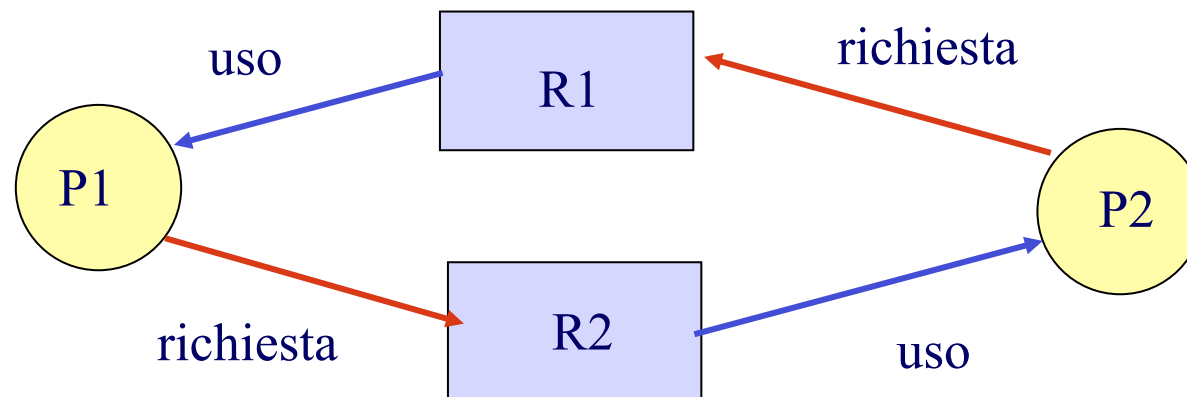
un insieme di processi è in *deadlock* se **ogni processo dell'insieme** è in attesa di un evento che può essere causato solo da un altro processo dell'insieme.

## Esempio

Si considerino due processi P1 e P2 che condividono le stesse risorse R1 e R2, da accedere in *mutua esclusione*; se contemporaneamente:

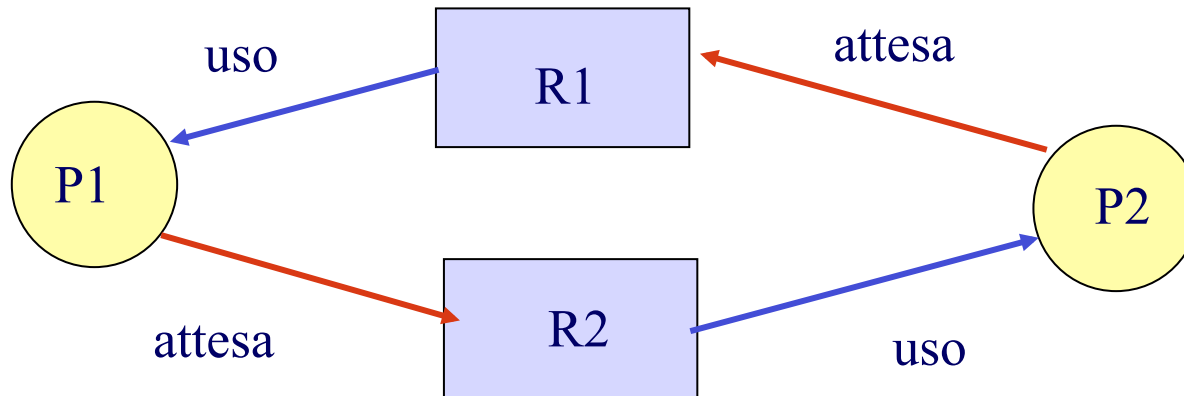
- P1 sta usando R1 e richiede R2
- P2 sta usando R2 e richiede R1

➔ P1 e P2 sono bloccati indefinitamente: **blocco critico!**



*Grafo di allocazione delle risorse*

# Deadlock



**P1:**

<richiesta R1>

<richiesta R2>●

<uso di R1 e R2>

<rilascio R2>

<rilascio R1>

**P2:**

<richiesta R2>

<richiesta R1>●

<uso di R1 e R2>

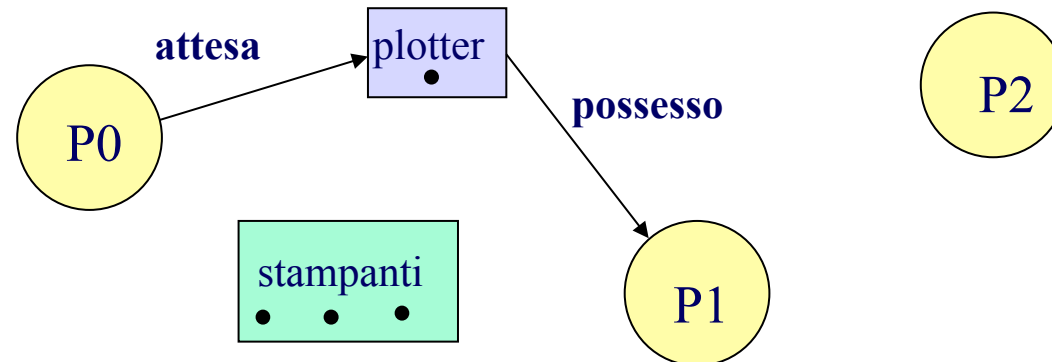
<rilascio R1>

<rilascio R2>

# Modello di riferimento

Per descrivere il deadlock, si utilizzano i grafi di allocazione delle risorse:

- processi
- risorse:
  - » classi (ad esempio: stampanti, dischi, etc.)
  - » istanze (stampante1, stampante2,...)
- archi:
  - » **risorsa** -> **processo** : allocazione (*possesso*)
  - » **processo** -> **risorsa** : *attesa* della risorsa

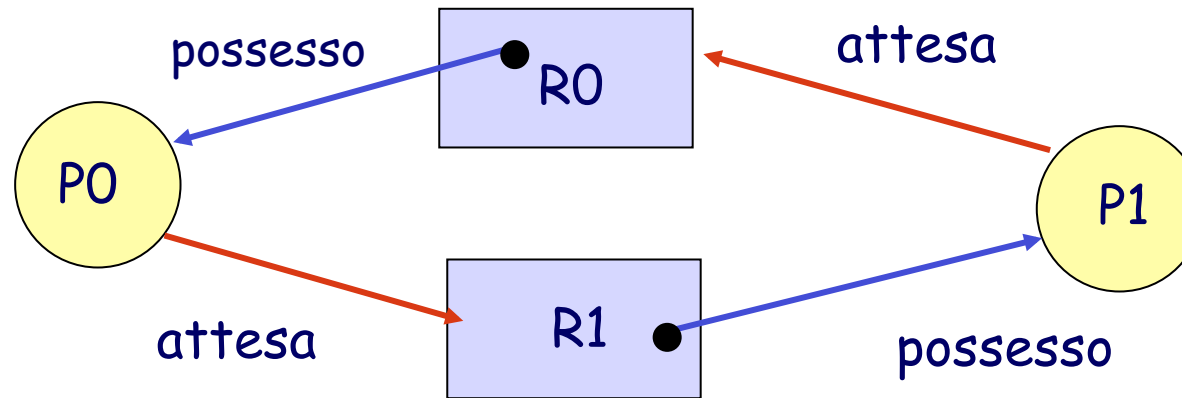


# Condizioni necessarie per il blocco critico

Dato un insieme di processi, essi si trovano in uno stato di *deadlock* se e solo se sono verificate le seguenti 4 condizioni:

- 1 **mutua esclusione**: le risorse sono utilizzate in modo mutuamente esclusivo.
  - 2 **possesso e attesa**: ogni processo che possiede una risorsa può richiederne un'altra.
  - 3 **impossibilità di prelazione**: una volta assegnata ad un processo, una risorsa non può essere sottratta al processo (*no preemption*).
  - 4 **attesa circolare**: esiste un gruppo di processi  $\{P_0, P_1..P_N\}$  in cui  $P_0$  attende una risorsa posseduta da  $P_1$ ,  $P_1$  attende una risorsa posseduta da  $P_2$ ,... e  $P_N$  attende una risorsa posseduta da  $P_0$ .
- **Condizioni necessarie e sufficienti**: se almeno una delle 4 condizioni non è verificata, non c'è *deadlock*!

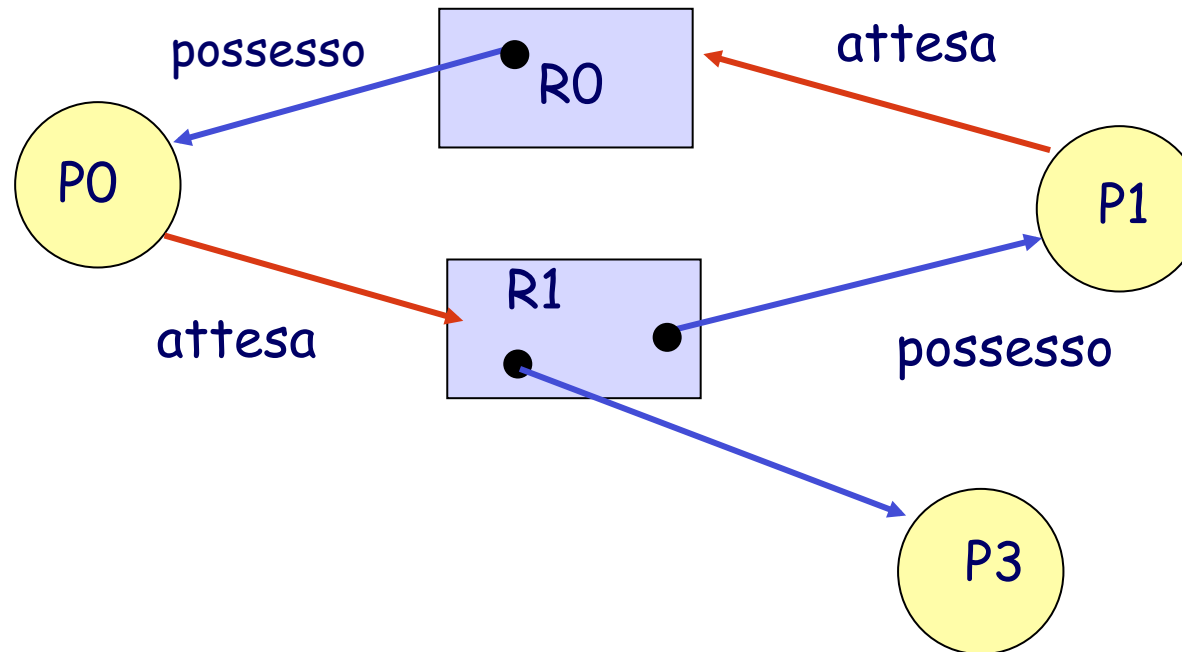
# Esempio



**HP:** mutua esclusione, no preemption

Se ogni risorsa ha una sola istanza, la presenza di un ciclo nel diagramma di allocazione, indica **deadlock**.

# Esempio



**HP:** mutua esclusione, no preemption, R1 ha 2 istanze.

-> in questo caso non c'è *deadlock*.



# Trattamento del blocco critico

Le situazioni di blocco critico vanno evitate o risolte.

## Approcci:

1. **prevenzione**: si evita *a-priori* il verificarsi di situazioni di blocco critico:
  - **prevenzione statica**: si impongono vincoli sulla struttura dei programmi, garantendo a priori che almeno una delle condizioni necessarie per il deadlock non sia verificata
  - **prevenzione dinamica** (*deadlock avoidance*): la prevenzione è attuata in modo dinamico: quando un processo P richiede una risorsa R, il sistema operativo verifica se l'allocazione di R a P può portare a situazioni di blocco critico:
    - » possibilità di deadlock (sequenza non salva): attesa del processo.
    - » altrimenti (sequenza salva): allocazione della risorsa R a P.
2. **rilevazione/ripristino del deadlock**: non c'è prevenzione, ma il S.O. **rileva** la presenza di deadlock -> algoritmo di **ripristino**

# Prevenzione statica

Si impone che almeno 1 delle 4 condizioni necessarie non sia verificata:

1. **mutua esclusione**: utilizzare risorse condivisibili (in certi casi non è possibile, dipende dalla natura della risorsa)
2. **possesso e attesa**:
  - si impedisce ad ogni processo di possedere una risorsa mentre ne richiede un'altra (uso di una risorsa alla volta)
  - richiesta di tutte le risorse necessarie nella stessa fase:
    - » complessità dell'operazione di acquisizione
    - » scarso utilizzo delle risorse
3. **preemption**: possibilità di sottrarre la risorsa al processo (in certi casi non è possibile)
4. **attesa circolare**: si stabilisce un rigido ordinamento nell'acquisizione delle risorse da parte di ogni processo:

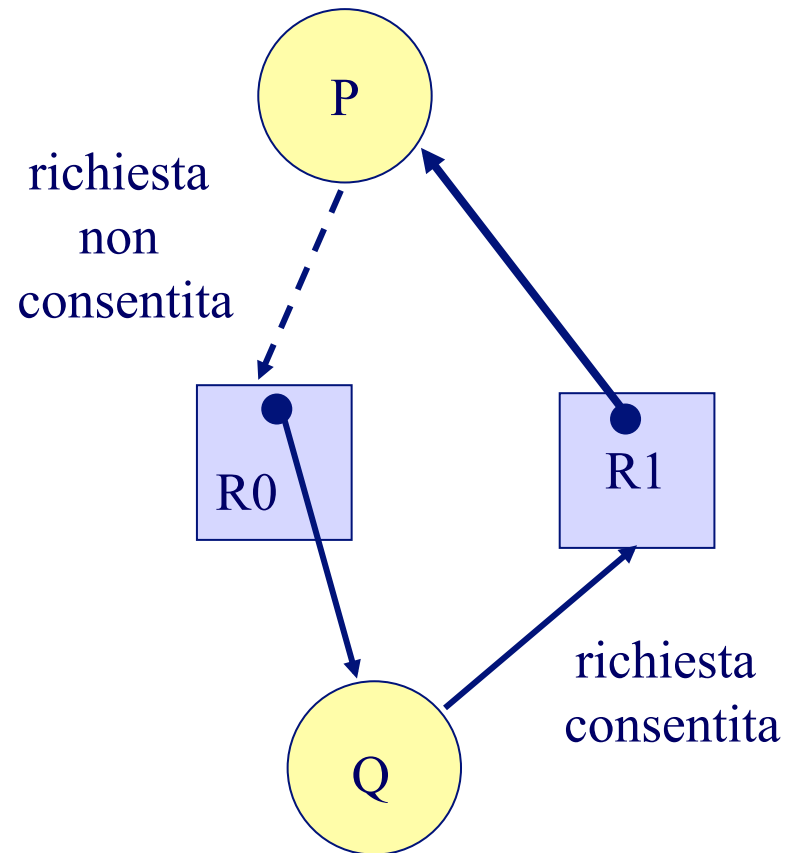
**Processi** = { $P_1, P_2, \dots, P_N$ }

**Risorse** = { $R_1, R_2, \dots, R_M$ }

- ogni processo  $P_i$  non può acquisire  $R_j$  se è già in possesso di  $R_k$  ( $k > j$ )

# Prevenzione statica

- **Allocazione gerarchica delle risorse:**  
ogni processo non può acquisire  $R_j$  se è già in possesso di  $R_k$  ( $k > j$ ).



## Prevenzione dinamica

Il sistema operativo previene il *deadlock*, mediante specifici algoritmi; ad esempio, l'*Algoritmo del Banchiere* (Dijkstra 1965):

- Un sistema, nell'allocare le risorse che vengono richieste, deve procedere come una **Banca**:
  - ▣ processi = clienti che possono richiedere credito (entro un limite dato)
  - ▣ risorse allocabili = soldi.
- La banca non può permettere contemporaneamente a tutti i clienti di ottenere il credito massimo: il sistema non potrebbe disporre di risorse a sufficienza => **Deadlock**

# Prevenzione dinamica

## A tempo di esecuzione:

quando un processo  $P$  richiede una risorsa  $R$ , il sistema operativo verifica se l'allocazione di  $R$  a  $P$  può portare a situazioni di blocco critico.

## Analisi delle possibili sequenze di esecuzione:

una sequenza di processi  $\{P_0, P_1, \dots, P_n\}$  è salva se per ogni processo  $P_i$  le richieste di risorse che  $P_i$  può ancora effettuare possono essere soddisfatte con le risorse attualmente libere, più le risorse allocate a tutti i processi  $P_j$  ( $j < i$ ).

# Prevenzione dinamica: esempio

Consideriamo 3 processi {P0, P1, P2} che competono per l'uso di 12 risorse (ad esempio, risorse dello stesso tipo):

	necessità max	uso attuale	possibili richieste
P0	10	5	5
P1	4	2	2
P2	9	2	7

-> rimangono 3 risorse disponibili

Lo stato attuale è uno stato salvo, perchè esiste una **sequenza salva**:  
{P1, P0, P2}

- ✓ P1 può terminare con le 3 risorse attualmente disponibili
- ✓ P0 può terminare con le attuali più quelle possedute da P1
- ✓ P2 può terminare con le attuali + quelle di P1 + quelle di P0

# Prevenzione dinamica: esempio

	necessità max	uso attuale	possibili richieste
P0	10	5	5
P1	4	2	2
P2	9	2	7

-> rimangono 3 risorse disponibili

## Sequenza salva:{P1, P0, P2}:

- P1 può terminare con le 3 risorse attualmente disponibili:  
P1 acquisisce 2 risorse:      usate=11      disp=1  
P1 rilascia le risorse:      usate=7      disp=5
- P0 può terminare con le 5 risorse ora disponibili:  
acquisisce 5 risorse:      usate=12      disp=0  
rilascia le risorse:      usate=2      disp=10
- P2 può terminare con le 10 risorse ora disponibili.

# Prevenzione dinamica

Il sistema operativo previene il *deadlock*, mediante specifici algoritmi (ad esempio, l'*Algoritmo del Banchiere*).

# Rilevazione/ripristino

se il sistema operativo non ha meccanismi di prevenzione, il *deadlock* può essere gestito *a posteriori* mediante **rilevazione** e **ripristino**.

- ▣ **Rilevazione**: algoritmi che individuano situazioni di deadlock già in atto.
- ▣ **Ripristino**: soluzione del blocco a posteriori; possibili approcci:
  - » **terminazione** forzata di processi coinvolti (*rollback*)
  - » **prelazione** di risorse allocate a processi in deadlock

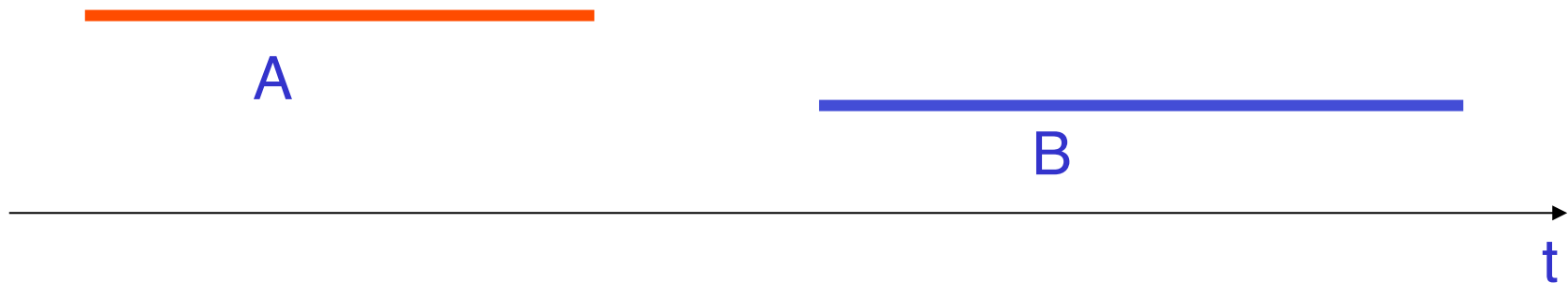
**Problema:** soluzione costosa



# Il Problema della Mutua Esclusione

# Mutua Esclusione

- Il problema della mutua esclusione nasce quando più di un processo alla volta può aver accesso a variabili/risorse comuni.
- La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle variabili comuni *non si sovrappongano nel tempo*.
- Nessun vincolo è imposto *sull'ordine* con il quale le operazioni sulle variabili vengono eseguite.



# Esempio di mutua esclusione

## Esempio

- Due processi P1 e P2 hanno accesso ad una struttura *organizzata a pila* rispettivamente per inserire e prelevare dati.
- La struttura dati è rappresentata da un *vettore **stack*** i cui elementi costituiscono i singoli dati e da una *variabile **top*** che indica la posizione dell'ultimo elemento contenuto nella pila.
- I processi utilizzano le operazioni ***Inserimento e Prelievo*** per depositare e prelevare i dati dalla pila.

```
typedef ... item;  
item stack[N];  
int top=-1;
```

```
void Inserimento(item y)  
{    top++;    stack[top]  
=y;  
}
```

```
item Prelievo()  
{    item x;  
    x= stack[top];  
    top--;  
    return x;  
}
```

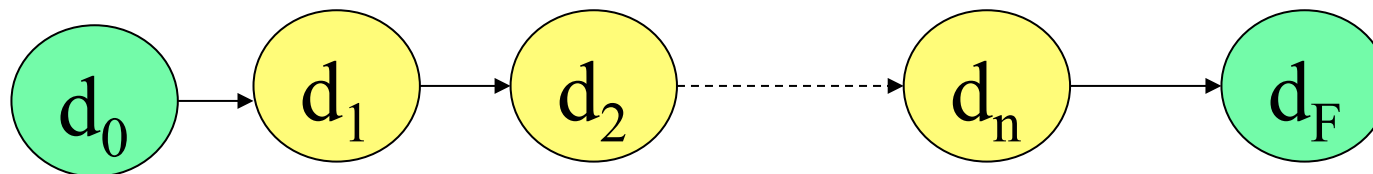
- L'esecuzione contemporanea di queste operazioni da parte dei processi può portare ad un uso scorretto della risorsa.
- Consideriamo questa sequenza di esecuzione:

T0:	<code>top++;</code>	(P1)
T1:	<code>x=stack[top];</code>	(P2)
T2:	<code>top--;</code>	(P2)
T3:	<code>stack[top]=y;</code>	(P1)

- Viene assegnato a x un valore *non definito* e l'ultimo valore valido contenuto nella pila *viene cancellato* dal nuovo valore di y.
- Potremmo individuare situazioni analoghe, nel caso di esecuzione contemporanea di una qualunque delle due operazioni da parte dei due processi.
- Il problema sarebbe risolto se le due operazioni di Inserimento e Prelievo fossero eseguite sempre in **mutua esclusione** (istruzioni indivisibili).

# Istruzioni Indivisibili

- Data un'istruzione  $I(d)$ , che opera su un dato  $d$ , essa e' **indivisibile** (o **atomica**) , se, durante la sua esecuzione da parte di un processo  $P$ , il dato  $d$  non e' accessibile ad altri processi.
  - $I(d)$ , a partire da un valore iniziale  $d_0$  , puo' operare una serie di trasformazioni sul valore di  $d$ , fino a giungere allo stato finale  $d_F$ ;
- ➔ poiche'  $I(d)$  e' indivisibile, gli stati intermedi non possono essere rilevati da altri processi concorrenti.



# Sezione Critica

- La sequenza di istruzioni con le quali un processo accede e modifica un insieme di variabili comuni prende il nome di **sezione critica**.
- Ad un insieme di variabili comuni possono essere associate *una sola sezione critica* (usata da tutti i processi) o *più sezioni critiche* (**classe di sezioni critiche**).
- La **regola di mutua esclusione** stabilisce che:  
*Sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo.*  
**oppure**  
*Una sola sezione critica di una classe può essere in esecuzione ad ogni istante.*

# Requisiti della soluzione al problema della mutua esclusione

Una corretta soluzione al problema della mutua esclusione deve soddisfare i seguenti requisiti:

1. Sezioni critiche della stessa classe devono essere eseguite in modo *mutuamente esclusivo*.
2. Quando un processo si trova all'esterno di una sezione critica *non può rendere impossibile* l'accesso alla stessa sezione (o a sezioni della stessa classe) ad altri processi.
3. Assenza di *deadlock*.



# Schema Generale

- Ogni processo prima di entrare in una sezione critica deve chiedere l'autorizzazione eseguendo una serie di istruzioni che gli garantiscono *l'uso esclusivo della risorsa*, se questa è libera, oppure *ne impediscono l'accesso* se questa è già occupata (**PROLOGO**) .
- Al completamento dell'azione il processo deve eseguire una sequenza di istruzioni per dichiarare libera la sezione critica (**EPILOGO**)

**IPOTESI:** l'hardware garantisce la mutua esclusione *solo a livello di lettura e scrittura di una singola parola di memoria.*

# MUTUA ESCLUSIONE:

## Analisi di alcune soluzioni algoritmiche

### Soluzione 1: Disabilitazione delle interruzioni durante le sezioni critiche:

- **Prologo:** disabilitazione delle interruzioni
- **Epilogo:** abilitazione delle interruzioni

```
/* struttura processo: */
main()
{
    ...
    <disabilitazione delle interruzioni>;
    <sezione critica A>;
    <abilitazione delle interruzioni>;
    ...
}
```

## Problemi:

- La soluzione è *parziale* in quanto è valida solo per sezioni critiche che operano sullo stesso processore.
- *Elimina* ogni possibilità di concorrenza.
- Rende *insensibile* il sistema ad *ogni stimolo esterno* per tutta la durata di qualunque sezione critica

**Soluzione 2:** (*A, B*) classe di sezioni critiche,  
*libero* variabile logica, inizializzata al valore  
*true*, associata a tale classe:

`int libero=1;`

```
/* processo P1: */
main()
{
    ...
    while (!libero);

    libero=0;
    <sez. critica A>;
    libero=1;

    ...
}
```

```
/* processo P2: */
main()
{
    ...
    while (!libero);

    libero=0;
    <sez. critica B>;
    libero=1;

    ...
}
```

- La soluzione **non soddisfa** la proprietà di mutua esclusione nell'esecuzione delle sezioni critiche.

### Esempio:

T0 : P1 esegue l'istruzione **while** e trova *libero* = 1

T1 : P2 esegue l'istruzione **while** e trova *libero* = 1

T3 : P1 pone *libero*=0 ed entra nella sezione critica

T4 : P2 pone *libero*=0 ed entra nella sezione critica

➔ entrambi i processi si trovano contemporaneamente nella sezione critica!

**Soluzione 3:** alla classe di sezioni critiche (A,B..) viene associata la variabile *turno* che può assumere i valori 1 e 2 ed inizializzata a 1.

```
int turno=1;
```

```
/* processo P1: */  
main()  
{  
    ...  
    while (turno!=1);  
    <sezione critica A>;  
    turno=2;  
    ...  
}
```

```
/* processo P2: */  
main()  
{  
    ...  
    while (turno!=2);  
    <sezione critica B>;  
    turno=1;  
    ...  
}
```

- La soluzione assicura che un solo processo alla volta può trovarsi nella sezione critica.
- Essa tuttavia impone un **vincolo di alternanza** nella esecuzione delle sezioni critiche (violazione del requisito 2).
- Ad esempio:
  - ❑ se *turno* = 2, il processo P1 non può entrare nella sua sezione critica, anche se questa non è occupata da P2.
  - ❑ Solo quando P2 avrà eseguito la sezione critica B, P1 potrà eseguire la propria.

**Soluzione 4:** Alla classe di sezioni critiche (A,B,..) vengono associate due variabili logiche busy1 e busy2 inizializzate al valore *false* (0):

```
int busy1=0;  
int busy2=0;
```

```
/* processo P1: */  
main()  
{  
    ...  
    busy1=1;  
    while (busy2!=0);  
    <sezione critica A>;  
    busy1=0;  
    ...  
}
```

```
/* processo P2: */  
main()  
{  
    ...  
    busy2=1;  
    while (busy1!=0);  
    <sezione critica B>;  
    busy2=0;  
    ...  
}
```



- La soluzione assicura che **un solo processo alla volta** può trovarsi in una delle sezioni critiche.
- E' eliminato l'inconveniente della soluzione 2) in quanto la variabile *busy* associata ad un processo mantiene il valore *false* per tutto il tempo in cui il processo rimane all'esterno della sua sezione critica.
- Possono presentarsi situazioni in cui, a seconda della velocità relativa dei processi, questi **non possono entrare** nella loro sezione critica, pur essendo tali sezioni libere :
  - To : P1 pone *busy1* = 1;
  - T1 : P2 pone *busy2* = 1;
- P1 e P2 ripetono indefinitamente l'esecuzione di **while** senza poter entrare nelle rispettive sezioni critiche.

**=> Deadlock !**

## Soluzione 5: Algoritmo di Dekker (1965)

E' stata la prima soluzione corretta al problema della mutua esclusione

⇒ Soddisfa i requisiti 1, 2 e 3.

```
int busy1 =0;  
int busy2 =0;  
int turno=1; /*dominio {1,2}*/
```

- Il valore iniziale di *turno* è indifferente.

```
int busy1 =0;
int busy2 =0;
int turno=1; /*dominio {1,2}*/
```

```
/* processo P1: */
```

```
main()
```

```
{    ...
```

```
    busy1=1;
```

```
    while (busy2)
```

```
        if (turno==2)
```

```
        {    busy1=0;
```

```
            while (turno!=1) ;
```

```
            busy1=1;
```

```
        }
```

```
    <sezione critica A>;
```

```
    turno=2;
```

```
    busy1=0;
```

```
    ...
```

```
}
```

```
/* processo P2: */
```

```
main()
```

```
{    ...
```

```
    busy2=1;
```

```
    while (busy1)
```

```
        if (turno==1)
```

```
        {    busy2=0;
```

```
            while (turno!=2) ;
```

```
            busy2=1;
```

```
        }
```

```
    <sezione critica B>;
```

```
    turno=1;
```

```
    busy2=0;
```

```
    ...
```

```
}
```

# Commenti

- L'algoritmo di Dekker è stato inizialmente proposto per risolvere la mutua esclusione tra 2 processi [1965].

## *Starvation:*

*È un fenomeno che si verifica quando un processo attende un tempo infinito per l'acquisizione di una risorsa.*

- *Un algoritmo di mutua esclusione è starvation-free, se ad ogni processo che intende accedere ad una sezione critica è garantito l'accesso in un tempo finito.*
- In questa versione (2 processi) è stato dimostrato che nell'algoritmo di Dekker non è possibile che si verifichi Starvation.

# Generalizzazione al caso di $n$ processi

- Successivamente, l'algoritmo di Dekker è stato generalizzato al caso di  $N$  processi; varie proposte; ad esempio:
  - ▣ Dijkstra [1965]
  - ▣ Martin [1986]
  - ▣ ...
- È possibile dimostrare che queste "generalizzazioni" non sono starvation-free.

## Soluzione 6: Algoritmo di Peterson (1981)

- Le variabili utilizzate per la sincronizzazione sono:

```
int busy1 =0;  
int busy2 =0;  
int turno=1; /*dominio {1,2}*/
```

```
int busy1=0;
int busy2=0;
int turno=1; /*dominio {1,2}*/
```

```
/* processo P1: */
main()
{ ...
  busy1=1;
  turno=2;
  while(busy2 && turno==2);
  <sezione critica A>;
  busy1=0;
  ...
}
```

```
/* processo P2: */
main()
{ ...
  busy2=1;
  turno=1;
  while(busy1 && turno==1);
  <sezione critica B>;
  busy2=0;
  ...
}
```

# Attesa attiva

- Tutte le soluzioni viste realizzano l'attesa dei processi con cicli:
  - ▣ l'attesa consiste nella ripetizione del corpo del ciclo fino a che la condizione di ripetizione fallisce.
- Durante l'attesa ogni processo usa inutilmente la CPU: **attesa attiva** (**busy waiting**)



# Requisiti della soluzione al problema della mutua esclusione

1. Sezioni critiche della stessa classe devono essere eseguite in modo *mutuamente esclusivo*.
2. Quando un processo si trova all'esterno di una sezione critica *non può rendere impossibile* l'accesso alla stessa sezione (o a sezioni della stessa classe) ad altri processi.
3. Assenza di *deadlock*

## In aggiunta (requisiti opzionali):

4. Assenza di *starvation*
5. Inoltre, per un utilizzo efficiente della CPU, devono essere eliminate forme di *attesa attiva* (*busy waiting*) sospendendo l'esecuzione di un processo per tutto il tempo in cui non può avere accesso alla sezione critica.

# Soluzioni hardware

- Nelle soluzioni precedenti si è supposto che l'hardware garantisca la mutua esclusione *solo a livello di lettura e scrittura di una singola parola di memoria*.
- ➔ *L'indivisibilità è garantita soltanto:*
  - nella lettura di una singola variabile
  - nella scrittura (assegnamento) di singola variabile
- Molti processori prevedono istruzioni che consentono di *esaminare e modificare* il contenuto di una parola in un unico ciclo (es: `test_and_set`).
- In questo caso è possibile dare una *semplice soluzione* al problema della mutua esclusione.

# Istruzione test and set

è un'istruzione macchina che consente la lettura e la modifica di una parola in memoria in modo *indivisibile*, cioè in un *solo ciclo di memoria*.

```
int test-and-set(int *a)
{
    int R;
    R=*a;
    *a=0;
    return R;
}
```

Tramite questa istruzione è possibile realizzare a livello hardware il meccanismo lock/unlock per la mutua esclusione.

# Lock e Unlock

Sia  $x$  una variabile logica associata ad una classe di **sezioni critiche** inizializzata al valore 1 (*libera*):

$x=0$  risorsa occupata,

$x=1$  risorsa libera.

Definiamo su  $x$  le seguenti operazioni:

```
void lock(int *x)
```

```
{   while (!*x) ;
```

```
    *x=0 ;
```

```
}
```

```
void unlock(int *x)
```

```
{   *x=1 ;
```

```
}
```

# Lock e Unlock

- Soluzione al problema della mutua esclusione:

```
int x=1;
```

```
/* processo P1: */  
main()  
{ ...  
  lock (&x) ;  
  <sezione critica A>;  
  unlock (&x) ;  
  ...  
}
```

```
/* processo P2: */  
main()  
{ ...  
  lock (&x) ;  
  <sezione critica B>;  
  unlock (&x) ;  
  ...  
}
```

Si noti che a differenza della **lock**, l'operazione **unlock** è **indivisibile**.

**Ipotesi:** lock(x) e unlock(x) operazioni indivisibili.

- L'esecuzione contemporanea di due lock(x) (ciascuna su un diverso elaboratore) viene automaticamente sequenzializzata dall'hardware.
- I requisiti 1,2,3 sono *soddisfatti*.
- Il soddisfacimento del requisito 4, *non è implicito* nella soluzione. Per superare l'inconveniente della *starvation* occorre un'opportuna realizzazione del *meccanismo di arbitraggio* per l'accesso in memoria.
- Il requisito 5 *non è soddisfatto*, essendo presente nella lock una forma di ***attesa attiva***

# Indivisibilita` dell'operazione lock

## Istruzione test and set (x):

```
int test-and-set(int *a)
{
    int R;
    R=*a;
    *a=0;
    return R;
}
```

## → Realizzazione dell'operazione lock(x):

```
void lock(int *x)
{
    while (!test-and-set(x)) ;
}
```

# Implementazione di lock e unlock

Se il set di istruzioni dell'architettura prevede la test-and-set (ts1):

**lock(x) :**

```
ts1 register, x
cmp register, 1
jne lock
ret
```

(copia x nel registro e pone x=0)  
(il contenuto del registro vale 1?)  
( se x=0 ricomincia il ciclo)  
(ritorna al chiamante)

**unlock(x) :**

```
move x, 1
ret
```

(inserisce 1 in x)  
(ritorna al chiamante)



# Proprietà della soluzione basata su lock e unlock

- Si può applicare in ambiente *multiprocessore*.
- Attesa attiva:
  - ▣ accettabile nel caso di sezioni critiche *molto brevi*

# I SEMAFORI

# Semaforo

Un semaforo è uno **strumento di sincronizzazione** che consente di risolvere **qualsiasi problema di sincronizzazione** tra thread nel modello ad ambiente globale.

Generalmente è realizzato dal nucleo del sistema operativo.

# Semaforo: definizione

Un semaforo è un dato astratto rappresentato da un *intero non negativo*, a cui è possibile accedere *solo* tramite le due operazioni **p** e **v**, definite nel seguente modo:

**p(s) :**

```
while (!s) ;  
s--;
```

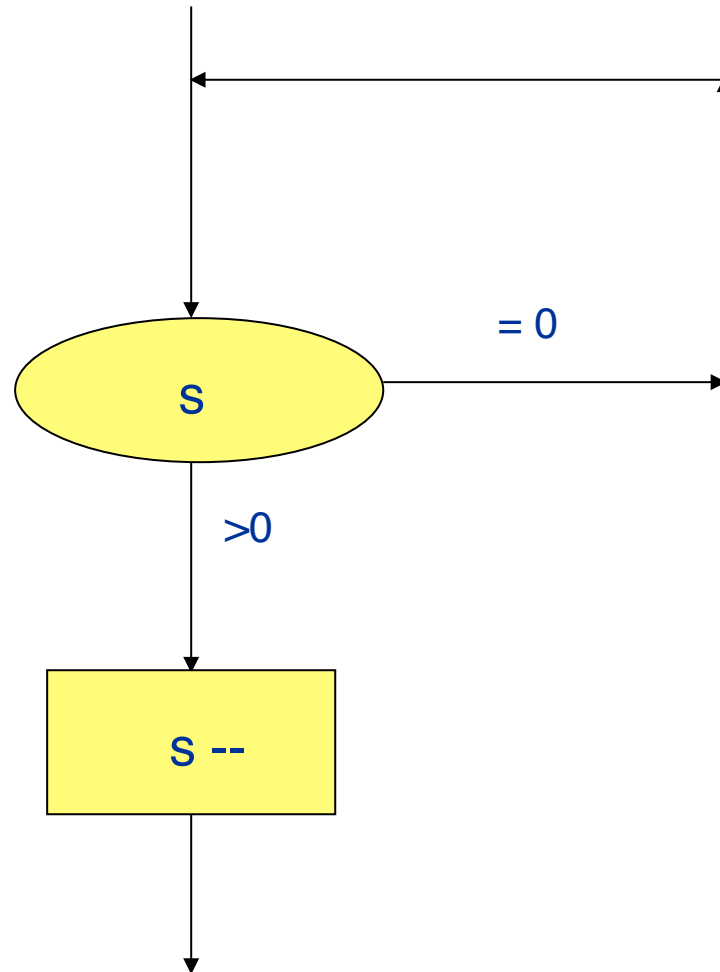
**v(s) :**

```
s++;
```

- L'operazione **p** ritarda il processo fino a che il valore del semaforo diventa maggiore di 0 e quindi decrementa tale valore di 1.
- L'operazione **v** incrementa di 1 il valore del semaforo.
- Le due operazioni sono **atomiche**. Il valore del semaforo viene modificato da un *solo processo* alla volta.

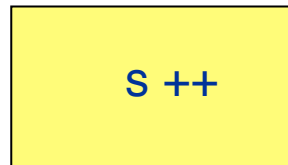
$p(s)$

$p$



$v(s)$

**v**



# Semaforo



Dato un semaforo  $S$ :

- Se il valore di  $S$  è 0: l'esecuzione di  $p$  provocherà l'attesa del processo che la invoca (semaforo **ROSSO**)
- Se il valore di  $S$  è maggiore di 0: la chiamata di  $p$  provocherà il decremento di  $S$  e la continuazione dell'esecuzione (semaforo **VERDE**).

# Esempio: Soluzione al problema della mutua esclusione con i semafori

```
semaphore mutex;  
mutex.value=1;
```

P1

```
.  
  
p (&mutex) ;  
<Sezione critica>;  
v (&mutex) ;  
  
.
```

P2

```
.  
  
p (&mutex) ;  
<Sezione critica>;  
v (&mutex) ;  
  
.
```

- *mutex* semaforo (binario) di mutua esclusione (0,1), con valore iniziale uguale a 1.

**SPD:** Qualunque sia la sequenza di esecuzione dei processi, la soluzione è sempre corretta.



# Realizzazione dei semafori

Il meccanismo di implementazione del costruito semaforo deve consentire:

- **eliminazione** di ogni forma di *attesa attiva* dei processi (v. definizione della *p*): sospensione del processo che non può proseguire l'esecuzione in una **coda** associata al semaforo.
- **eliminazione di forme di *starvation***: scelta **FIFO** del processo da risvegliare.

# Realizzazione dei semafori

Al semaforo sono associati:

- ▣ il valore intero `value` non negativo con valore iniziale  $\geq 0$
- ▣ una coda `Qs` nella quale sono posti i descrittori dei processi che attendono l'autorizzazione a procedere.

```
typedef struct{  
    unsigned int value;  
    queue Qs;} semaphore;
```

# Realizzazione di p e v

Le primitive p e v possono essere realizzate come segue:

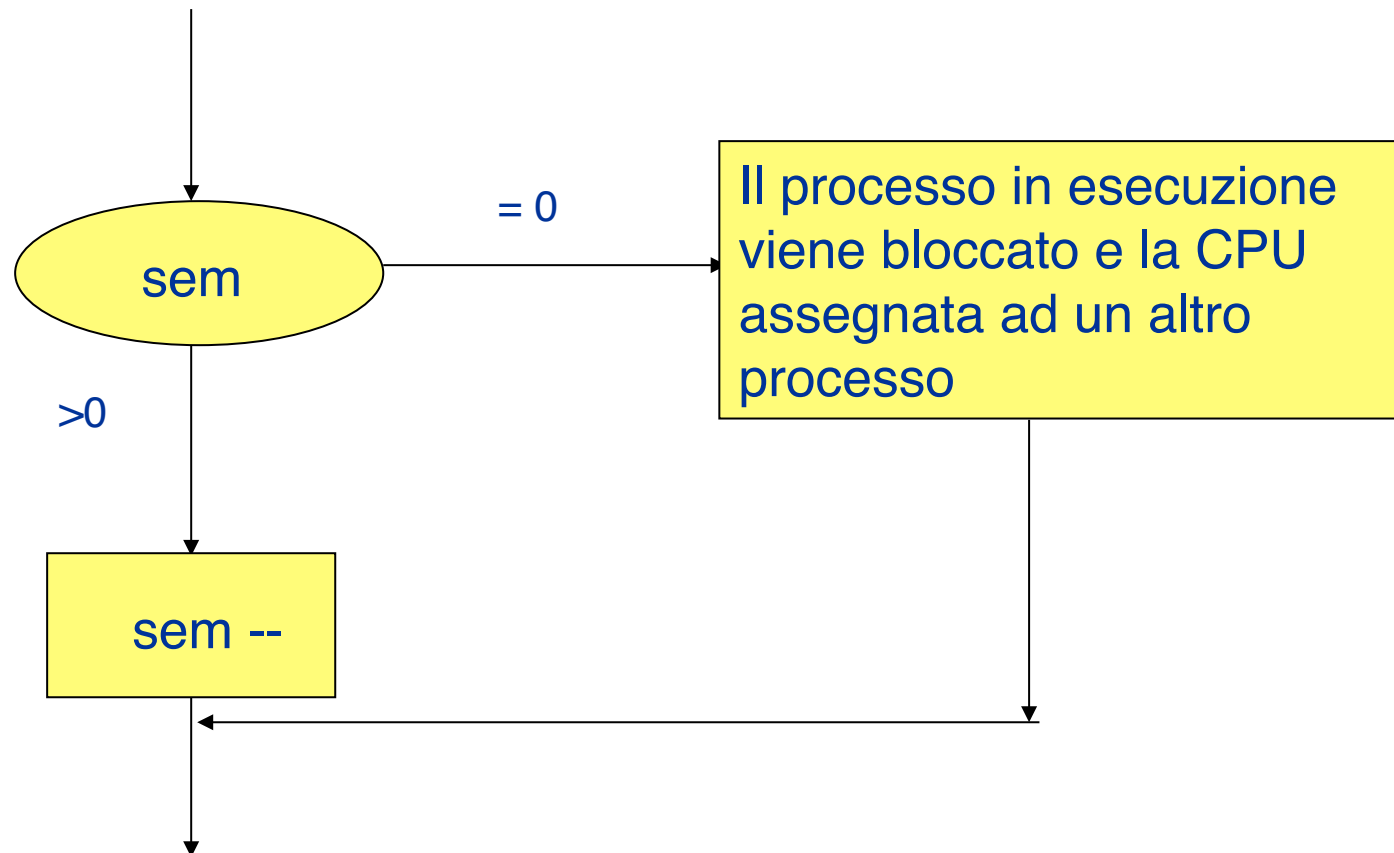
```
void p(semaphore *s) {  
    if (s->value==0)  
        <il processo viene sospeso ed il suo  
        descrittore inserito in s->Qs>  
    else s->value--;  
}
```

```
void v(semaphore *s) {  
    if (<s->Qs non e` vuota>)  
        <il descrittore del primo processo viene  
        rimosso dalla coda ed il suo stato  
        modificato in pronto>  
    else s->value++;  
}
```

- $p$  e  $v$  : sezioni critiche → devono essere **azioni indivisibili** (azioni atomiche):
  - uso di lock/unlock per imporre la mutua esclusione nell'esecuzione di  $p$  e  $v$ .
- Durante un'operazione sul semaforo nessun altro processo può accedere al semaforo fino a che l'operazione è completata o bloccata.

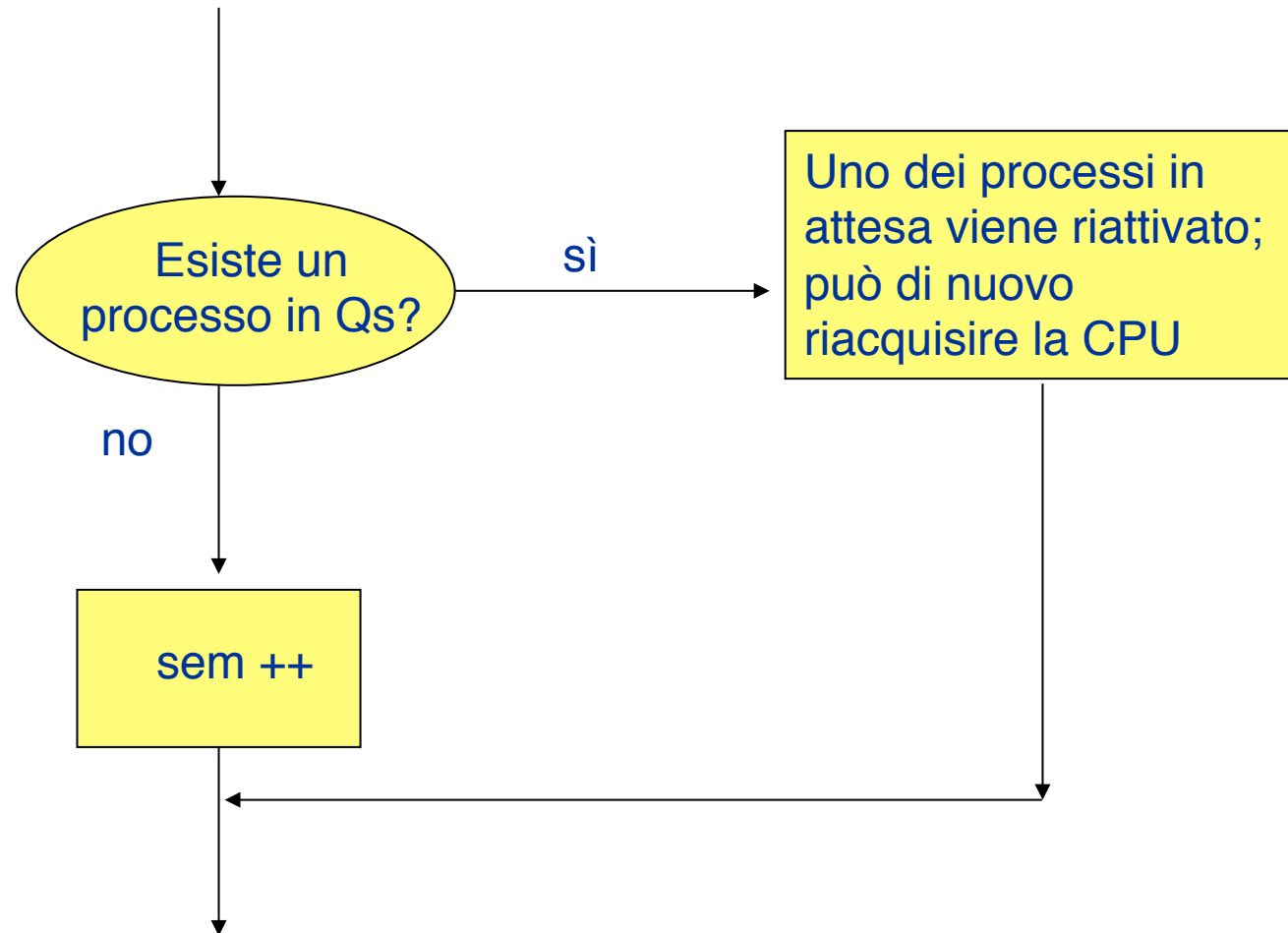
**p**

*p(sem)*



**V**

**v (sem)**



# Realizzazione semaforo: atomicità

```
typedef struct{ int value;  
               queue Qs;  
               int lock; // inizializzato a 1  
} semaphore;  
  
void p(semaphore *s) {  
    lock(s->lock);  
    if (s->value==0)  
    {    unlock(s->lock); //uscita sez. critica  
        <sospensione processo in s->Qs>  
        lock(s->lock); //entrata sez. critica  
    }  
    else s->value--;  
    unlock(s->lock);  
}
```

# Realizzazione semaforo: atomicità

```
void v(semaphore *s) {  
    lock(s->lock);  
    if (<s->Qs non e` vuota>)  
        <il descrittore del primo processo viene rimosso  
        dalla coda ed il suo stato modificato in pronto>  
    else s->value++;  
    unlock(s->lock);  
}
```



# Soluzione al problema della mutua esclusione con i semafori

```
semaphore mutex;  
mutex.value=1;
```

P1

```
      .  
p (&mutex) ;  
<Sezione critica>;  
v (&mutex) ;  
      .  
      .
```

P2

```
      .  
p (&mutex) ;  
<Sezione critica>;  
v (&mutex) ;  
      .  
      .
```

- **SPD**: Qualunque sia la sequenza di esecuzione dei processi, la soluzione è sempre corretta.
- *Mutex* viene chiamato "**semaforo di mutua esclusione**"; inizializzato a 1, potrà assumere soltanto i valori 0 e 1.

# Cooperazione tra processi concorrenti

- Scambio di **messaggi** generati da un processo e consumati da un altro
- Scambio di **segnali temporali** che indicano il verificarsi di dati eventi

*La cooperazione tra processi prevede che l'esecuzione di alcuni di essi risulti condizionata dall'informazione prodotta da altri (vincoli sull'ordinamento nel tempo delle operazioni dei processi).*

## ESEMPIO:

- n processi  $P_1, P_2, \dots, P_n$  attivati ad intervalli prefissati di tempo da  $P_0$ .
- l'esecuzione di  $P_i$  non può iniziare prima che sia giunto il segnale da  $P_0$
- ad ogni segnale inviato da  $P_0$  deve corrispondere una attivazione di  $P_i$

$n_1$ = numero di richieste di attivazione di  $P_i$

$n_2$ = numero di segnali di attivazione inviati da  $P_0$

$n_3$ = numero di volte in cui  $P_i$  è stato attivato

Deve essere ad ogni istante:

se  $n_2 \geq n_1$   $n_3 = n_1$

se  $n_2 < n_1$   $n_3 = n_2$

```
semaphore si;  
si.value=0  /* valore iniziale  $s_i = 0$ */
```

**processo  $P_i$  :**

```
main()
```

```
{ ...
```

```
  while(...)
```

```
  { ...
```

```
    p(&si) ;
```

```
    ...
```

```
  }
```

```
...  
}
```

**processo  $P_0$  :**

```
main()
```

```
{ ...
```

```
  while(...)
```

```
  { ...
```

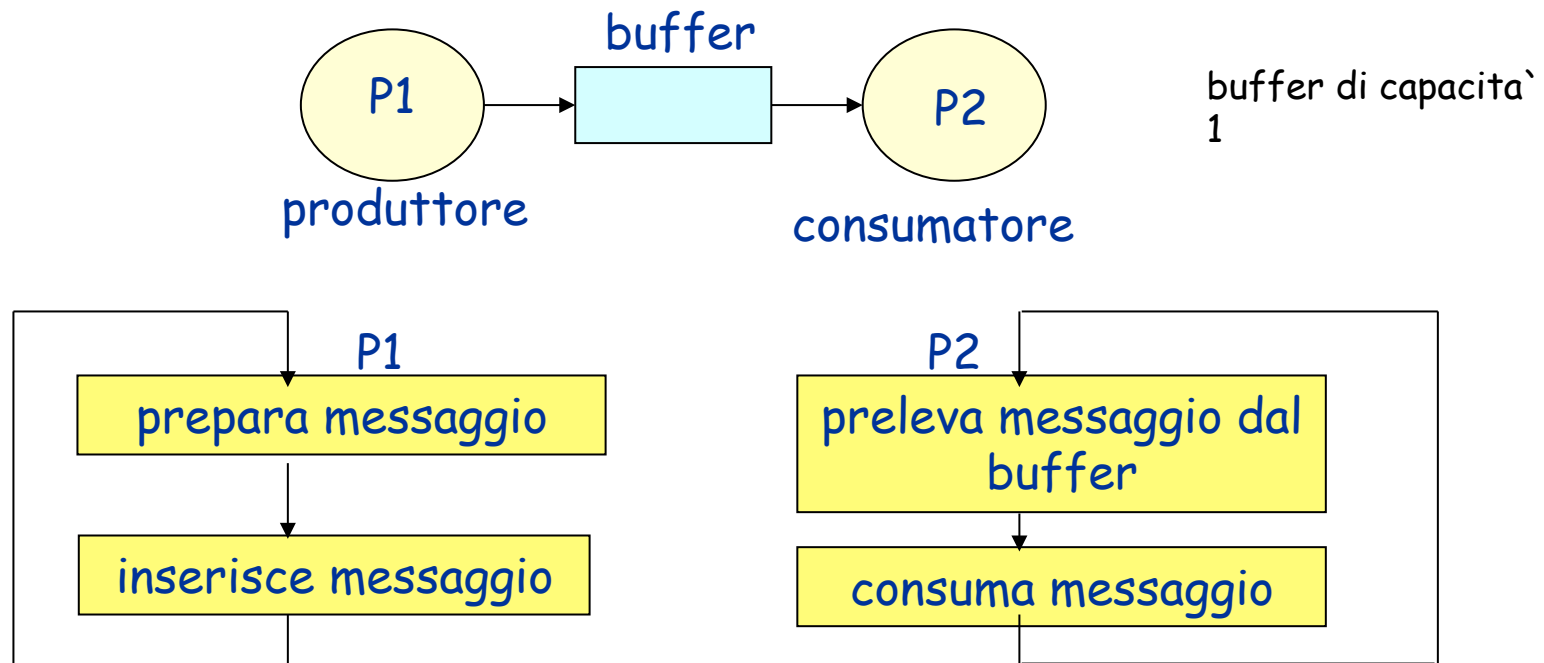
```
    v(&si) ;
```

```
    ...
```

```
  }
```

```
...  
}
```

# Comunicazione

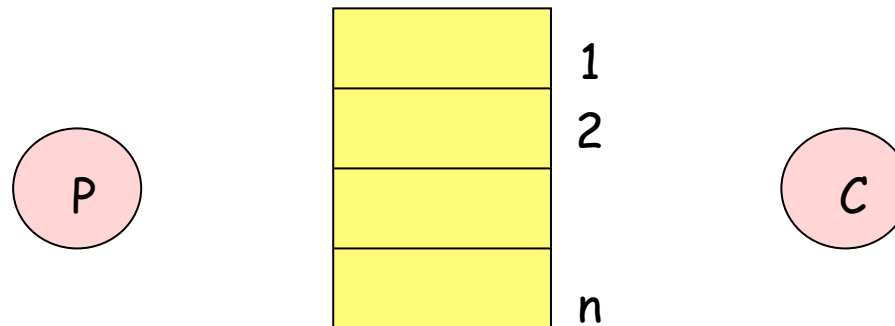


**Sequenza corretta:** inserimento-prelievo-  
inserimento-prelievo....

**Sequenze errate:**

- ❑ Inserimento-inserimento-prelievo....
- ❑ Prelievo-prelievo-inserimento

## Esempio: Produttore Consumatore (buffer di capacita' n)



1. Il produttore non può inserire un messaggio nel buffer se questo è pieno.
2. Il consumatore non può prelevare un messaggio dal buffer se questo è vuoto

```
semaphore msg_disponibile;  
msg_disponibile.value=0;
```

```
/* Processo produttore:*/  
main()  
{  
    for (;;) {  
        <produzione messaggio>;  
        <deposito messaggio>;  
        v(&msg_disponibile);  
    }  
}
```

```
/* Processo consumatore:*/  
main()  
{  
    for (;;) {  
        p(&msg_disponibile);  
        <prelievo messaggio>;  
        <consumo messaggio>;  
    }  
}
```

- questa soluzione soddisfa soltanto la condizione 2: il produttore potrebbe depositare un messaggio nel buffer pieno!

- Per sincronizzare correttamente gli accessi al buffer di produttore e consumatore, introduciamo *due semafori*:

- spazio\_disp (valore iniziale=n, elementi liberi nel buffer)
- msg\_disp (valore iniziale=0, num messaggi nel buffer)

```
semaphore spazio_disp, msg_disp;  
spazio_disp.value=n;  
msg_disp.value=0;
```

```
/* Processo produttore:*/  
main()  
{  
    for (;;) {  
        <produzione messaggio>;  
        p(&spazio_disp);  
        <deposito messaggio>;  
        v(&msg_disp);  
    }  
}
```

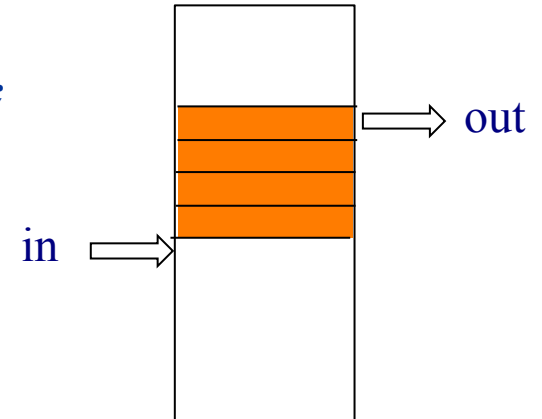
```
/* Processo consumatore:*/  
main()  
{  
    for (;;) {  
        p(&msg_disp);  
        <prelievo messaggio>;  
        v(&spazio_disp);  
        <consumo messaggio>;  
    }  
}
```

I due semafori si dicono “**semafori risorsa**”, poichè il valore di ognuno di essi rappresenta la disponibilità corrente delle risorse ad esso associate (elementi liberi, messaggi disponibili).



## Dettagli sulla gestione del Buffer

```
semaphore spazio_disp, msg_disp;  
spazio_disp.value=n;  
msg_disp.value=0;  
msg buffer[n]:  
int out=0, in=0;
```



```
/* Processo produttore:*/  
main()  
{ msg M;  
  for (;;) {  
    <produzione messaggio M>;  
    p(&spazio_disp);  
    buffer[in]=M;  
    in=(in + 1)%n;  
    v(&msg_disp);  
  }  
}
```

```
/* Processo consumatore:*/  
main()  
{ msg M  
  for (;;) {  
    p(&msg_disp);  
    M=buffer[out];  
    out=(out + 1)%n;  
    v(&spazio_disp);  
    <consumo messaggio M>;  
  }  
}
```

Nel caso di più produttori e più consumatori:

aggiungiamo i due semafori mutex1 e mutex2

...

```
semaphore mutex1, mutex2;  
mutex1.value=1;  
mutex2.value=1;
```

### Processo produttore:

```
main()  
{ for(;;)  
  { <produz. messaggio>;  
    p (&spazio_disp);  
    p (&mutex1);  
    <inserimento mess.>;  
    v(&mutex1)  
    v(&msg_disp);  
  }  
}
```

### Processo consumatore:

```
main()  
{ for(;;)  
  { p (&msg_disp);  
    p (&mutex2)  
    <prelievo mess.>;  
    v(&mutex2)  
    v(&spazio_disp);  
    <consumo messaggio>;  
  }  
}
```

# Esempi di uso dei semafori: gestione di risorse

- $R_1, R_2, \dots, R_n$   $n$  unità di uno stesso tipo di risorsa (tutte **equivalenti** fra loro).
- $P_1, P_2, \dots, P_m$   $m$  processi che devono operare su una qualunque risorsa in **modo esclusivo** tramite le operazioni  $A, B, \dots$

## I Soluzione

- Si assegna un semaforo di mutua esclusione  $M_i$  (v.  $i.=1$ ) ad ogni risorsa  $R_i$

processo  $P_s$ :

..

$p(M_i);$

$R_i.A;$

$v(M_i);$

..

$R_i.A$  rappresenta l'esecuzione  
dell'operazione  $A$  su  $R_i$

## Inconvenienti della soluzione:

- Come decide il generico processo su quali risorse operare (come viene scelto  $i$ )?
- Può capitare che, una volta scelta  $R_i$ , se su di essa sta operando in quel momento un secondo processo  $P_k$ , il processo  $P_s$  si blocchi su  $p(M_i)$ , pur essendo disponibili altre risorse  $R_h$  ( $h \neq j$ ).

**II Soluzione:** viene introdotta una nuova risorsa *G*, gestore di *R1, R2, ... Rn*. Essa può essere concepita come una **struttura dati** destinata a mantenere lo stato delle risorse gestite. Sul gestore si opera tramite due procedure:

***Richiesta e Rilascio.***

```
unsigned int Richiesta();
```

```
void Rilascio(unsigned int x);
```

*(dove il parametro *x* rappresenta l'indice della risorsa assegnata o rilasciata)*

## Strutture dati del gestore:

- le procedure *Richiesta* e *Rilascio* dovranno essere eseguite in **mutua esclusione**
- ➔ semaforo **mutex** di **mutua esclusione** con v.i. = 1
- Un processo che esegue *Richiesta* verifica la disponibilita` di una qualunque risorsa  $R_j$ .
- Un processo che esegue *Rilascio* rende nuovamente disponibile una risorsa
- ➔ semaforo **risorsa** **ris** con valore iniziale = n
- E' necessario un vettore di variabili *booleane* **Liberato** **[i]** per registrare quale risorsa è in un certo istante libera ( $\text{Liberato}[i] = 1$ ) e quale occupata ( $\text{Liberato}[i] = 0$ ).

## II Soluzione - segue

```
#define n 20

semaphore  mutex, ris;

int Libero[n];/* stato di allocazione risorse */

void inizializza()
{/*inizializzazione del gestore:*/
mutex.value= 1;
ris.value= n;
for(i = 0; i<n; i++)
    Libero[i] = 1; /*true*/
}
```

## II Soluzione - segue

```
int Richiesta ()
{
    unsigned int x, i;
    p(&ris);
    p(&mutex);
    i=0;
    do
        i++;
    while (! Libero[i]);
    x = i;
    Libero[i] = 0;
    v(&mutex);
    return x;
}
```

```
void Rilascio (unsigned int x)
{
    unsigned int i;
    p(&mutex);
    i=x;
    Libero[i]= 1;
    v(&mutex);
    v(&ris);
}
```



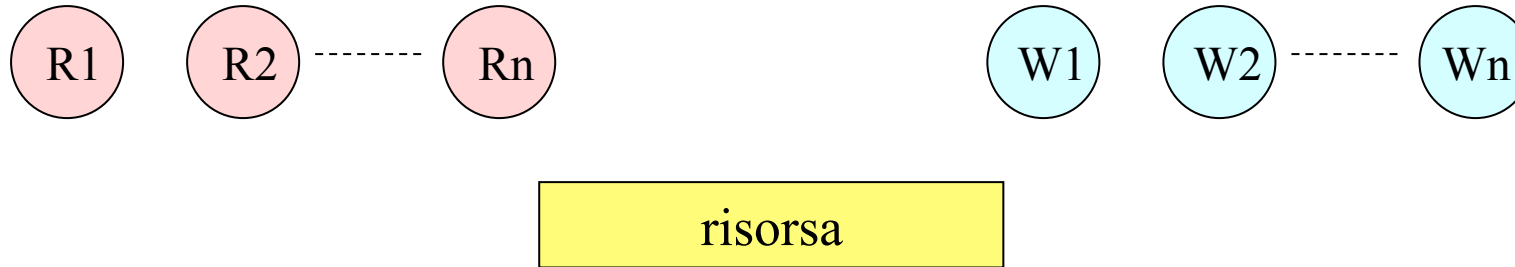
# Schema del processo:

```
void Processo()  
{  
    unsigned int r;  
    ...  
    r=Richiesta();  
    <uso della risorsa r>  
    Rilascio(r);  
    ...  
}  
main()  
{inizializza();  
    <creazione processi di codice Processo>  
    ...}
```

# Realizzazione di politiche di gestione delle risorse

- Nei problemi di sincronizzazione visti precedentemente si ha che:
  - ▣ La decisione se un processo possa proseguire l'esecuzione dipende dal valore di un solo semaforo (es., "mutex", "spazio disponibile", "messaggio disponibile")
  - ▣ La scelta del processo da riattivare avviene tramite l'algoritmo implementato nella  $v$  (FIFO).
- In problemi di sincronizzazione più complessi si ha che:
  - ▣ Decidere se un processo possa proseguire l'esecuzione dipende in generale dal verificarsi di una **condizione di sincronizzazione**
  - ▣ La scelta del processo da riattivare può avvenire in base a criteri diversi da quelli adottati nell'implementazione di  $v$  (ad es. sulla base di **priorità tra processi**).

# Esempio: Problema dei "readers and writers"



## Condizioni di sincronizzazione:

- I processi lettori possono usare la risorsa contemporaneamente.
- I processi scrittori hanno accesso esclusivo alla risorsa.
- I processi lettori e scrittori si **escludono mutuamente** nell'uso della risorsa.

## Soluzione

Un processo lettore aspetta solo se la risorsa è già stata assegnata ad un processo scrittore: cioè nessun lettore aspetta se uno scrittore è già in attesa (possibilità di attesa infinita da parte dei processi scrittori).

## Soluzione :

```
int    readcount=0;

semaphore mutex, w;

mutex.value=1; w.value=1;
```

### READER

```
reader()
{  p(&mutex);
  readcount ++;
  if (readcount == 1)
    p(&w);
  v(&mutex);
  ..
  <lettura>
  ..
  p(&mutex);
  readcount --;
  if (readcount==0)
    v(&w);
  v(&mutex);
}
```

### WRITER

```
writer()
{  p(&w);
  . .
  <scrittura>
  . .
  v(&w);
}
```

# Costrutti linguistici per la sincronizzazione

I semafori costituiscono un meccanismo **molto potente** per la sincronizzazione dei processi. Tuttavia, l'uso dei semafori può risultare troppo "a basso livello".

## → Possibilità di errori

**Esempio:** mutua esclusione

- *scambio tra p e v:*

```
v(mutex) ;  
<sez. critica>;  
p(mutex) ;
```

più processi possono operare nella sezione critica.

- *utilizzo erraneo di p e v:*

```
p(mutex) ;  
<sez. critica>;  
p(mutex) ; -> deadlock.
```

- *etc.*

# Costrutti linguistici per la sincronizzazione

→ Per ovviare a problemi di questa natura si sono introdotti costrutti linguistici di “più alto livello” :

- regioni critiche semplici e condizionali
- monitor