Il costrutto Monitor

Corso di Laurea in Ingegneria Informatica

Università degli Studi di Napoli Federico II

Anno Accademico 2024/2025, Canale San Giovanni

Il costrutto Monitor

- Sommario
  - Il tipo Monitor: struttura e strategie di controllo
  - Monitor signal and wait
  - Monitor signal and continue
- Riferimenti
  - Dispensa su costrutto monitor (Ancillotti – Boari, "Principi e Tecniche di Programmazione Concorrente")
  - Dispensa su costrutto monitor (W. Stallings, "Operating Systems : Internals and Design Principles")
  - www.ostep.org, Cap. 30 + Appendix D

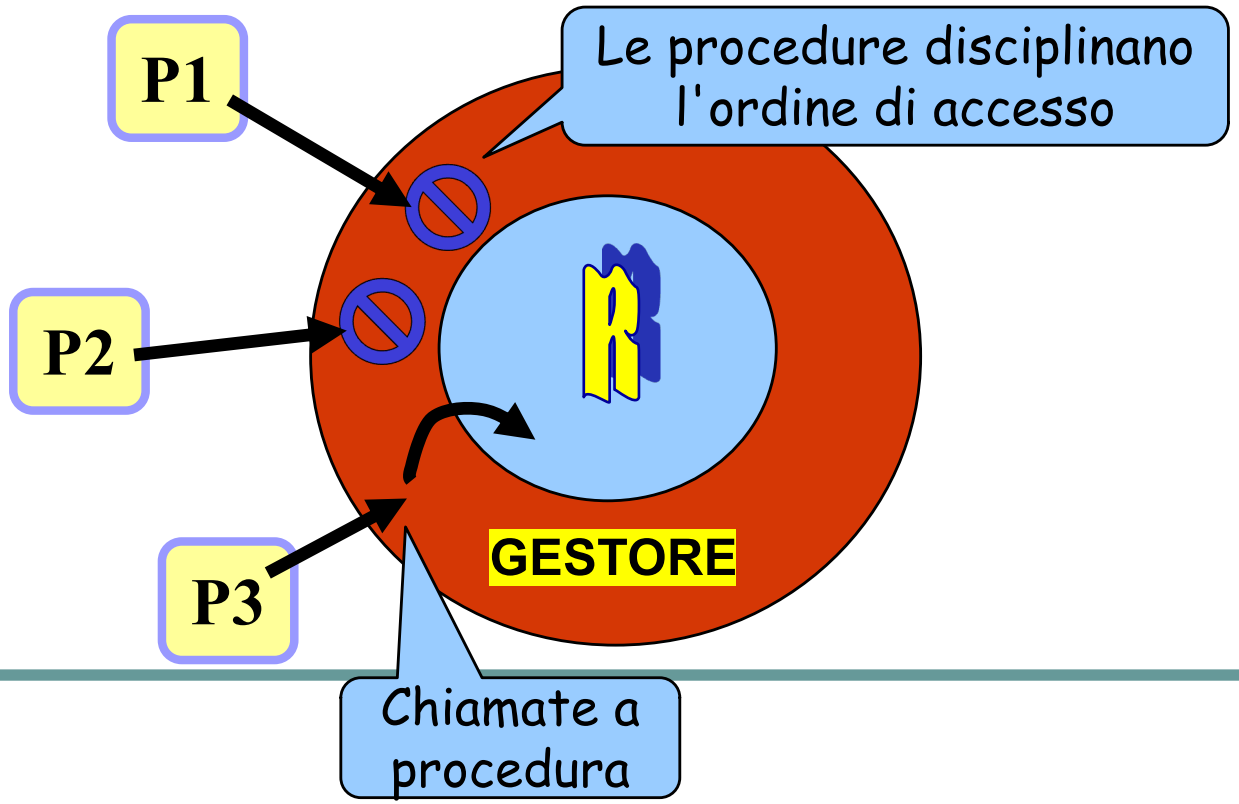
Introduzione

- Il monitor è un **costrutto sintattico**
- Abbina un insieme di **operazioni** ad una **struttura dati (risorsa) condivisa** tra processi

il costrutto Monitor è sintatticamente simile al costrutto **class** ...
...ma utilizzato per la gestione di **risorse condivise**

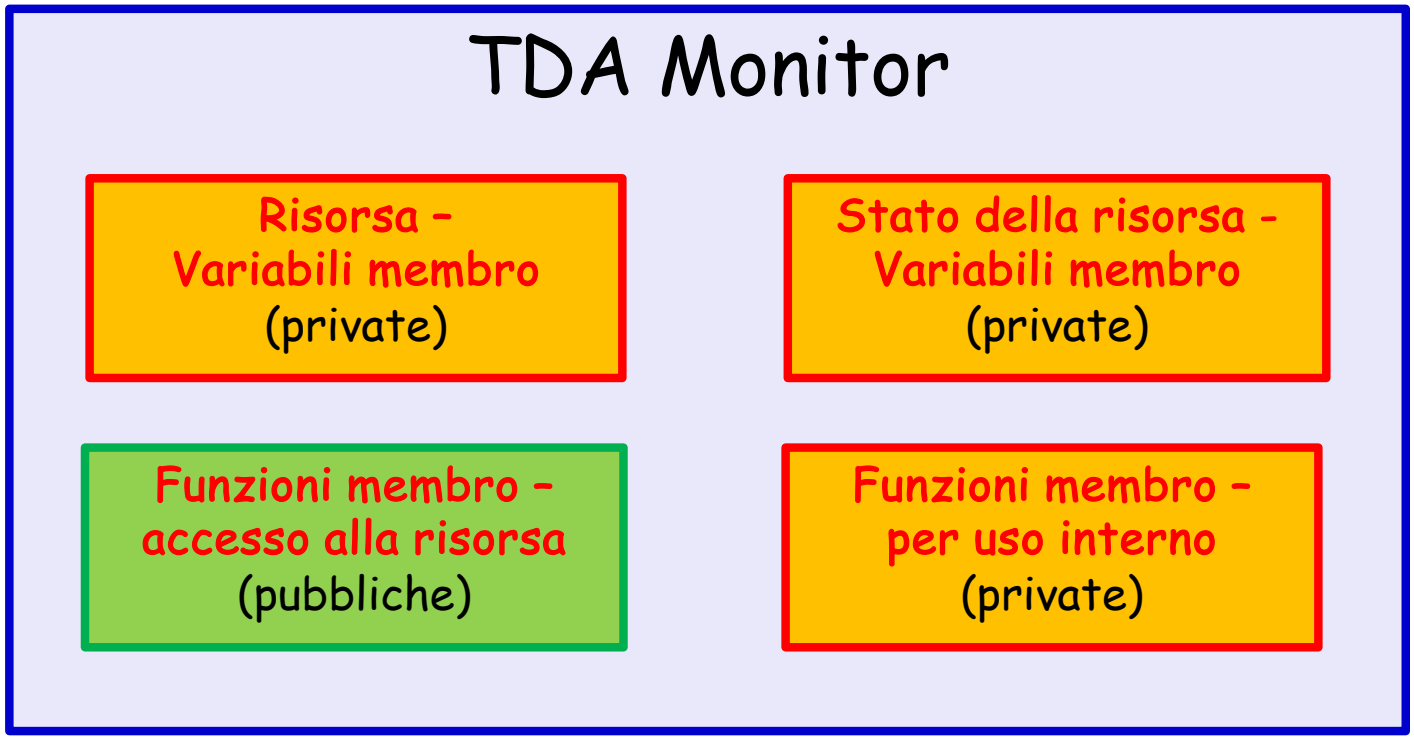
Introduzione

- Facilita la programmazione concorrente
- Permette di creare **politiche di accesso** alle risorse condivise



Tipo di dato astratto

- Si configura come «**tipo di dato astratto**»



Strategie di controllo

La politica di accesso impone che:

1. un solo processo alla volta può avere accesso alla risorsa condivisa (competizione)

2. i processi seguano un determinato ordine di accesso alla risorsa (cooperazione)

Strategie di controllo

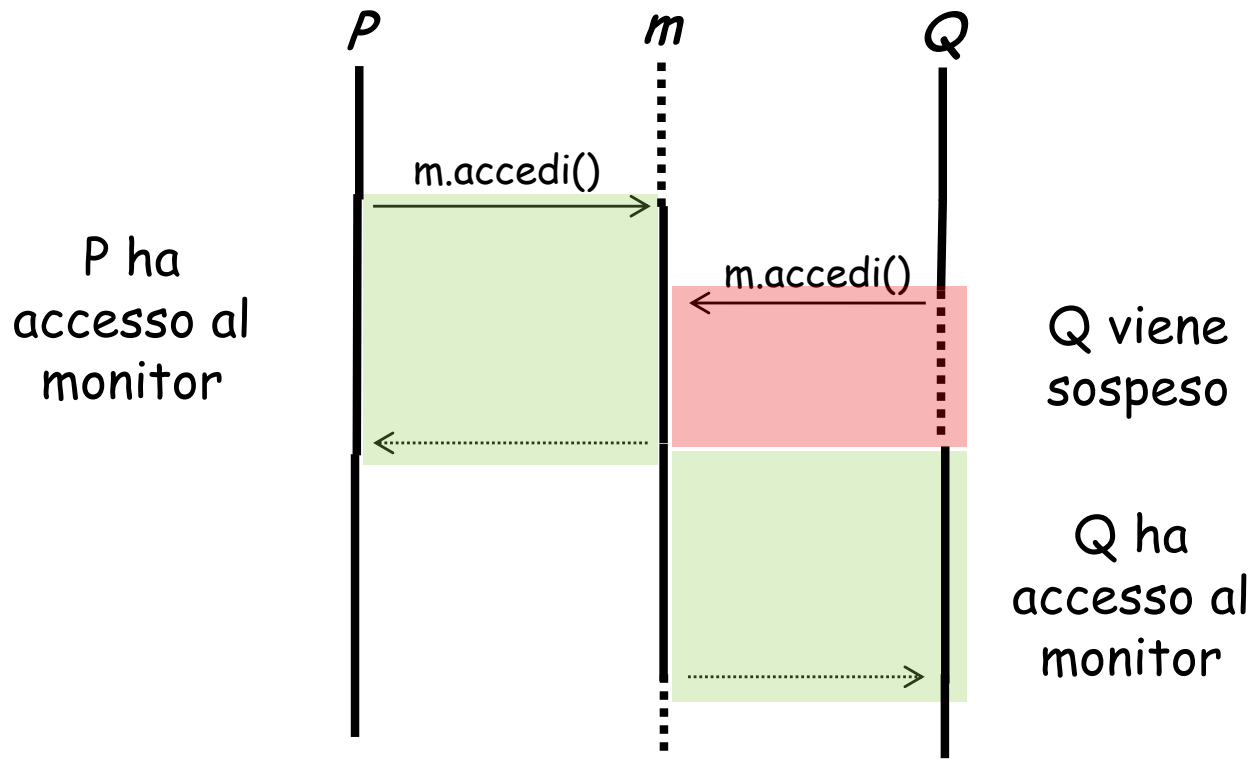
La politica di accesso impone che:

1. un solo processo alla volta può avere accesso alla risorsa condivisa (competizione)

2. i processi seguano un determinato ordine di accesso alla risorsa (cooperazione)

Le funzioni pubbliche del monitor sono eseguite in modo mutuamente esclusivo

Competizione



Competizione

- I **metodi pubblici** di un monitor si definiscono con:
  - **librerie di funzioni**
  - **parole chiave** del linguaggio di programmazione

```
Monitor M {
        void metodo1() {
                enter_monitor();
                // operazioni su risorsa
                leave_monitor();
        }
        ...
}
```

Strategie di controllo

La politica di accesso impone che:

1. un solo processo alla volta può avere accesso alla risorsa condivisa (competizione)

2. i processi seguano un determinato ordine di accesso alla risorsa (cooperazione)

I processi si sospendono se non è verificata una "condizione logica" di accesso

Cooperazione

- Per la sospensione, si introduce un tipo di variabile (interna al monitor), detta variabile condition

```
var_cond x;
```

- Definisce due metodi:
  - `x.wait_cond()` sospende del processo chiamante...
  - ....fino a che un altro processo esegue `x.signal_cond()`

Coo

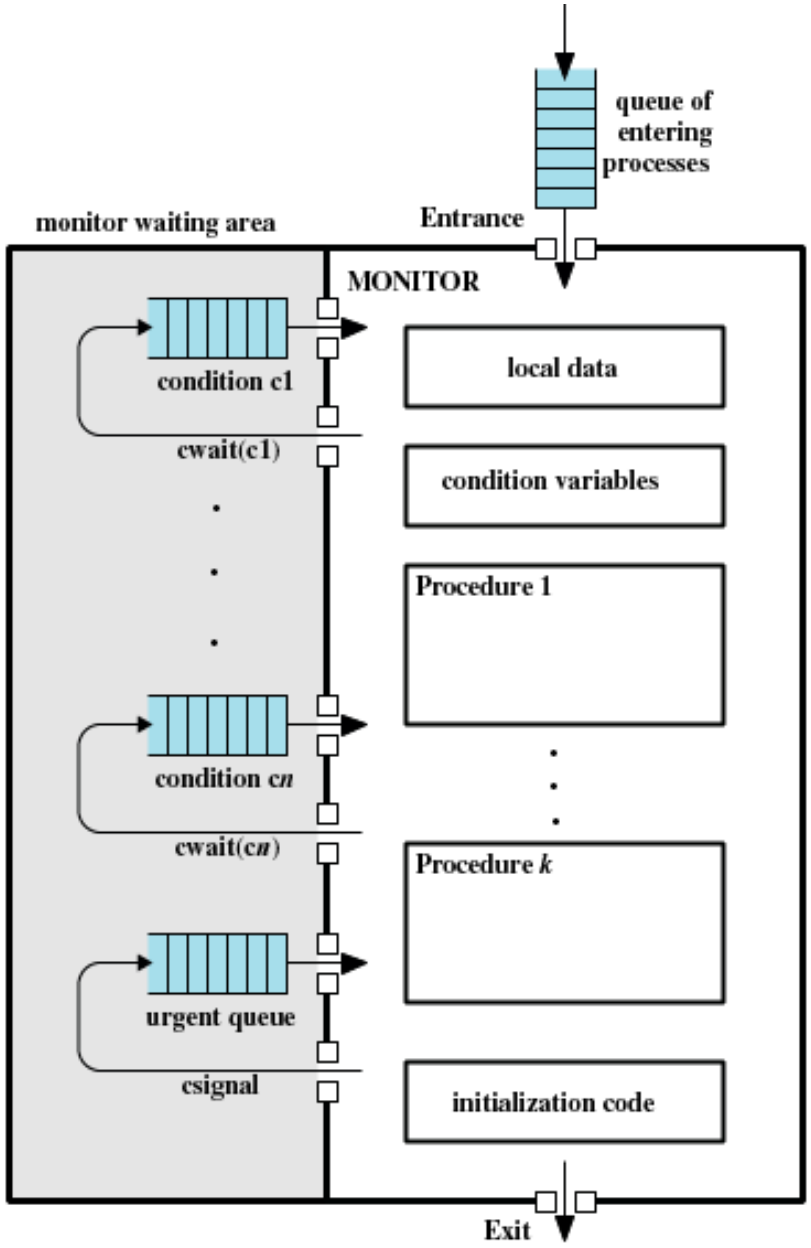
```
Monitor M {
        var_cond x;
        var_cond y;

        void metodo1() {
                enter_monitor();
                if/while (! condizione_logica) {
                        x.wait_cond();
                }
                // operazioni su risorsa …
                y.signal_cond();
                leave_monitor();
        }
        …
}
```

Cooperazione

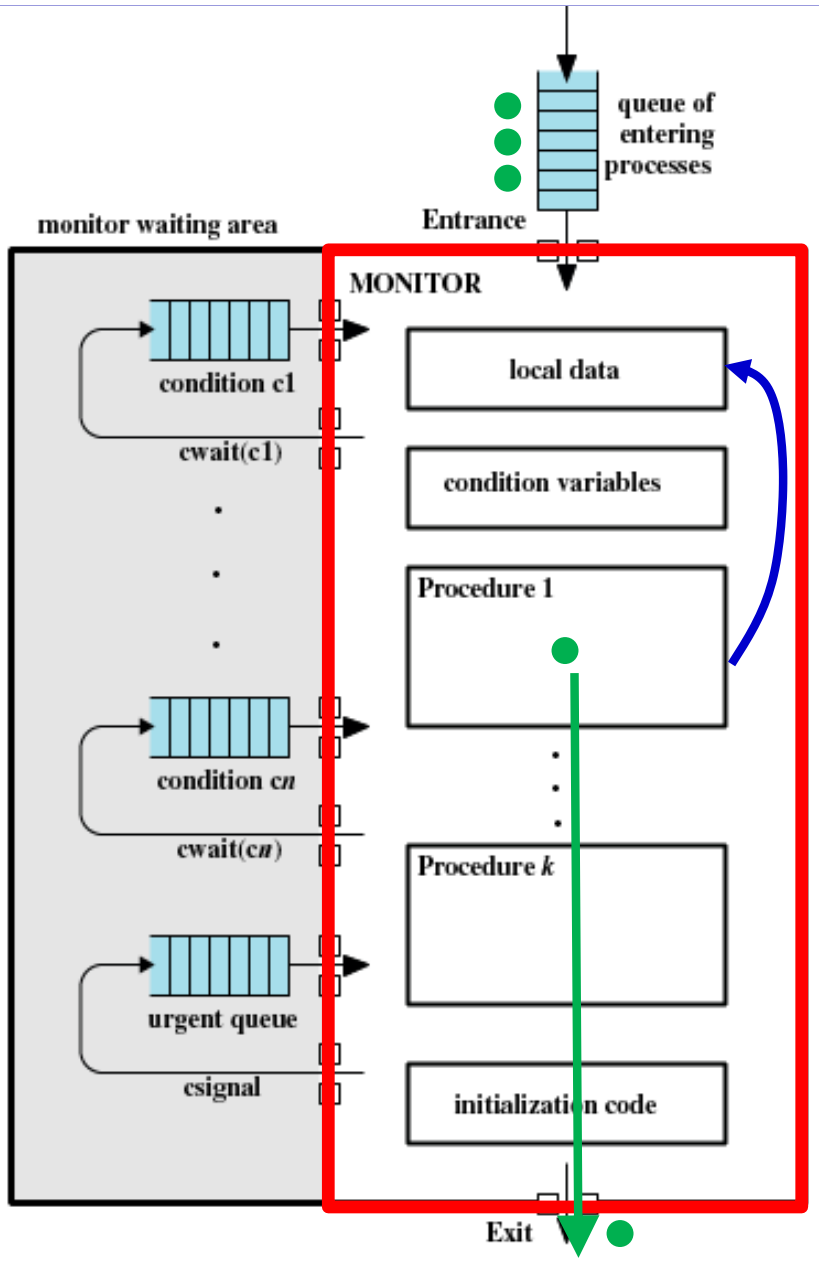
- Per ottenere un **ordine di accesso**:

1. in ogni metodo del monitor, il processo chiamante controlla se è soddisfatta una **condizione logica**
2. se la condizione **non è verificata**, il processo chiamante viene **sospeso**
3. si consente l'accesso ad un altro processo, che può eventualmente **risvegliare** il processo sospeso

Monitor - panoramica



Monitor - panoramica

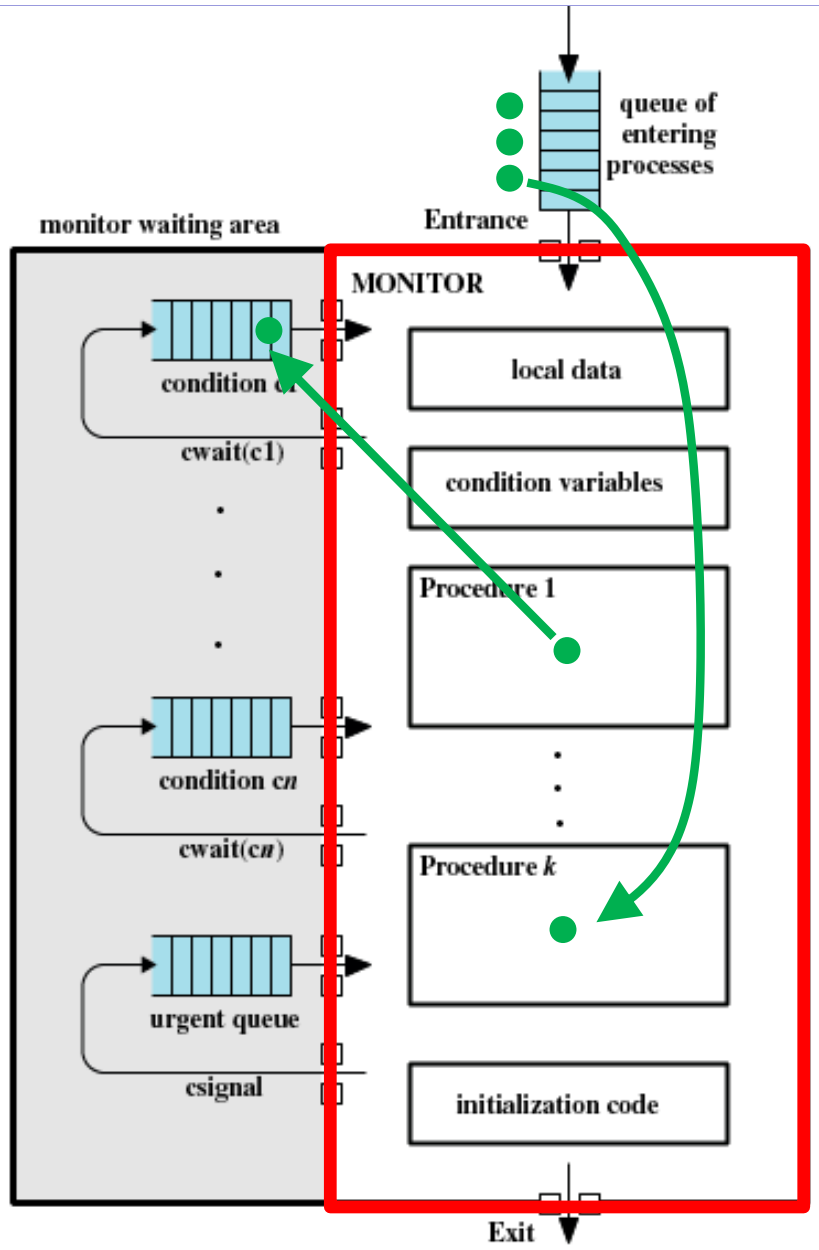


Le procedure sono eseguite in mutua esclusione, da al più un processo

Le procedure controllano, tramite le variabili locali, se la **condizione di sincronizzazione** è valida (es. "buffer vuoto")

Se la **condizione di sincronizzazione** è valida, il processo completa l'esecuzione e libera il monitor

Monitor - panoramica



Se al momento del controllo la **condizione di sincronizzazione** non è valida, il processo si pone volontariamente in attesa usando una delle condition variables

Mentre è in attesa, il monitor diventa libero, e si lascia accedere un altro processo

Note su variabili condition

var. condition $\neq$ semafori

wait_condition() $\neq$ wait_sem()

signal_condition() $\neq$ signal_sem()

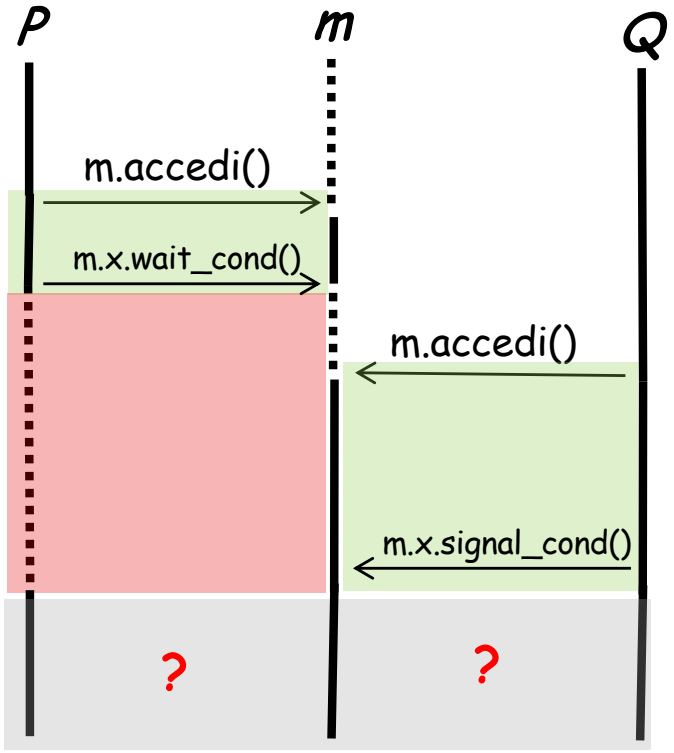
- La **wait_cond() sospende sempre il processo chiamante**
  - nei semafori, con wait_sem(), la sospensione era condizionata alla variabile intera del semaforo
- La **signal_cond()** non ha alcun effetto se non vi è alcun processo in attesa sulla variabile condition

Semantica dell'operazione signal

P ha accesso per primo al monitor...

...ma la risorsa **non è ancora pronta**.
P quindi **si sospende**.

P ritorna **pronto**.



Q ha accesso al monitor, e può **aggiornare lo stato della risorsa**

Quando la risorsa diventa **pronta**, il processo **Q riattiva P**

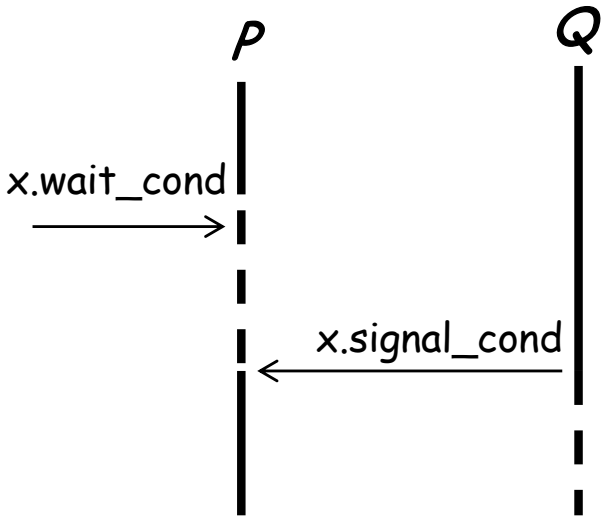
PROBLEMA:
P e Q non possono eseguire entrambi, si violerebbe la **mutua esclusione**!

Semantica dell'operazione signal

- Non esiste una soluzione unica!
- Diversi sistemi attribuiscono comportamenti (**semantica**) diversi alle primitive **wait_cond()**/**signal_cond()**

Prima soluzione: signal and wait

- *Signal_and_wait* prevede che
  - il processo segnalato P riprenda immediatamente l'esecuzione
  - il processo segnalante Q venga sospeso



Q viene sospeso per evitare che possa modificare nuovamente la condizione di sincronizzazione.

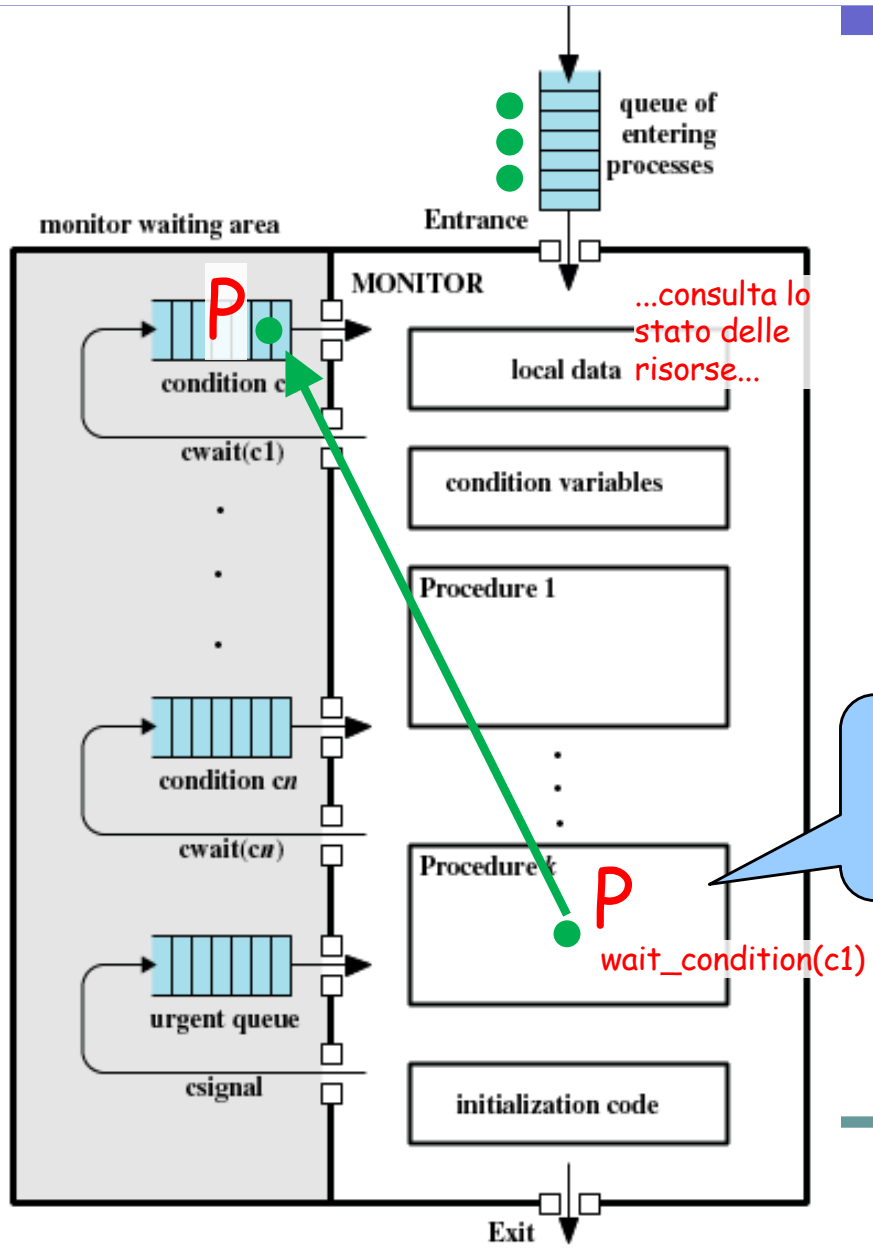
Uso della primitiva wait_cond

- In "signal and wait", il processo segnalato è il **primo ad eseguire**
- Al risveglio, il segnalato **ha certezza di trovare verificata** la condizione che attendeva

Lo schema tipico dell'invocazione di una *wait_cond* è all'interno di un «if»:

```
if (!B) {                    // B = condizione di sincronizzazione
    cv.wait_cond();          // cv = var. condition, abbinata a B
}
<...accesso alla risorsa...>
```

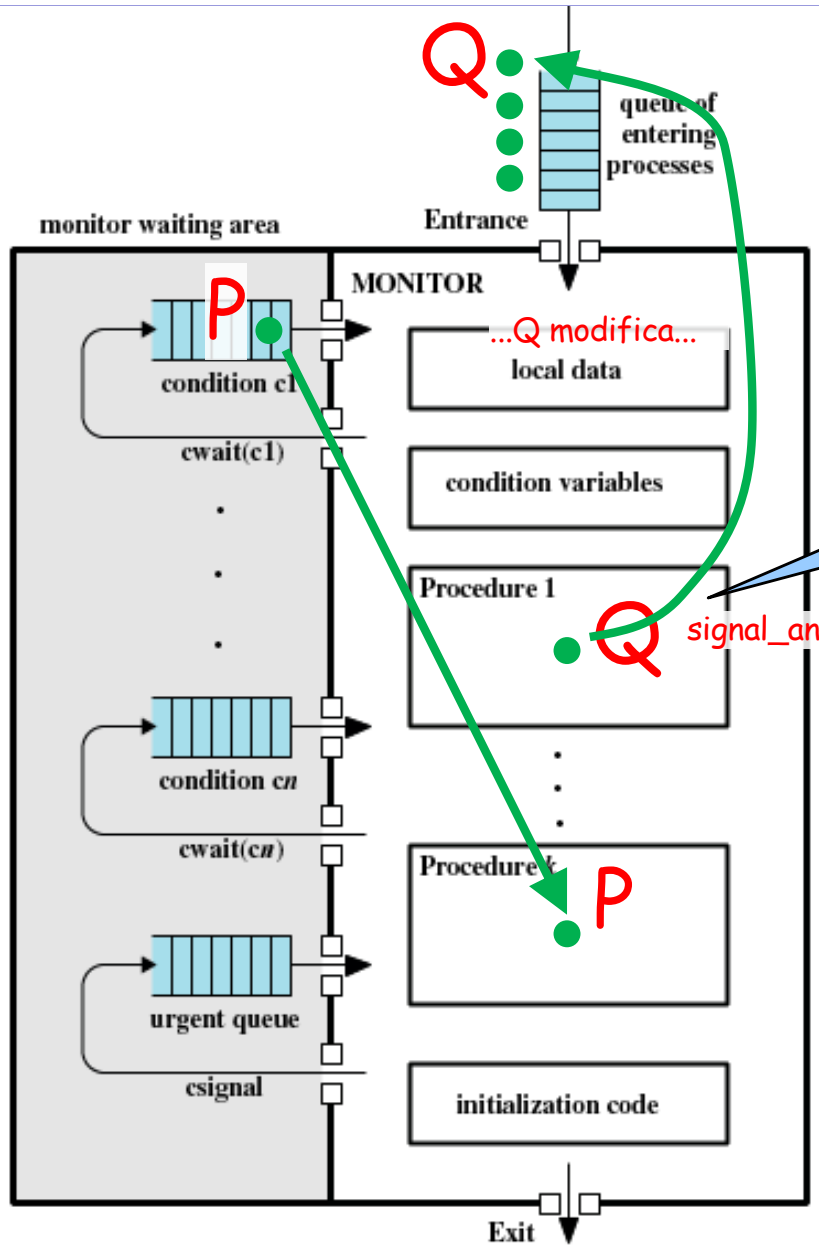
Monitor – signal and wait



Il processo P entra per primo, prematuramente (la condizione di sincronizzazione non è ancora valida), per cui si sospende

```
if(condizione non valida) {
    wait_condition(...)
}
```

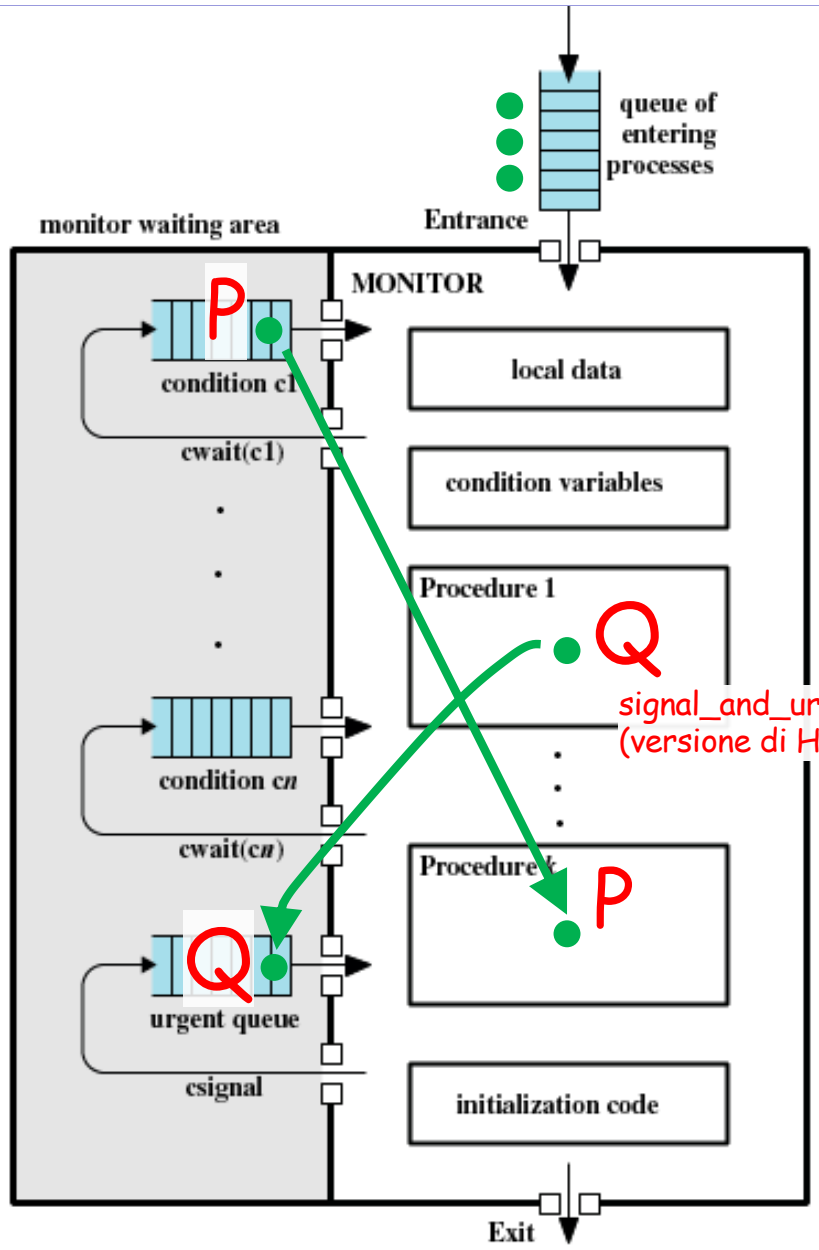
Monitor – signal and wait



<...opera sulla risorsa...>
signal_and_wait(...)

Il processo Q dovrà attendere l'uscita di P, e competere con altri processi per rientrare nel monitor

Monitor – soluzione di Hoare



Nella soluzione di Hoare, il processo Q ha la precedenza sugli altri processi che attendono di entrare nel monitor.

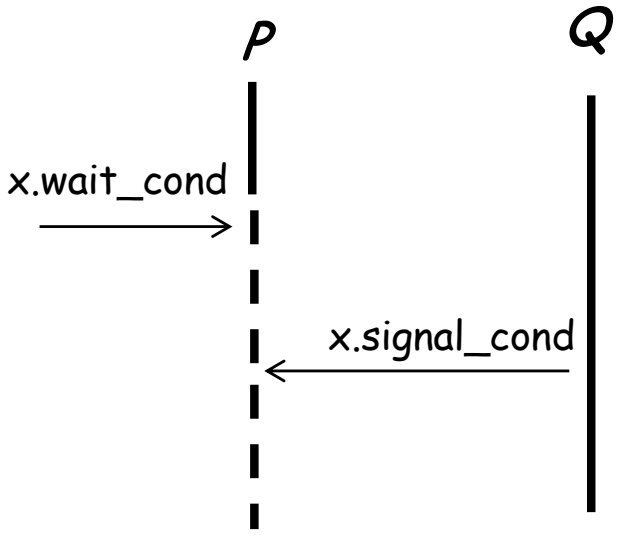
Si sospende il processo Q su un'apposita coda (*urgent_queue*), separata dal mutex

La soluzione di Hoare

- La soluzione di Hoare è un caso particolare di signal_and_wait, detta *signal_and_urgent_wait*
- Prevede che il processo Q abbia la **priorità** su ogni altro processo che intende entrare nel monitor
- Ciò si può ottenere sospendendo il processo Q su un'apposita coda (*urgent_queue*), separata dal mutex

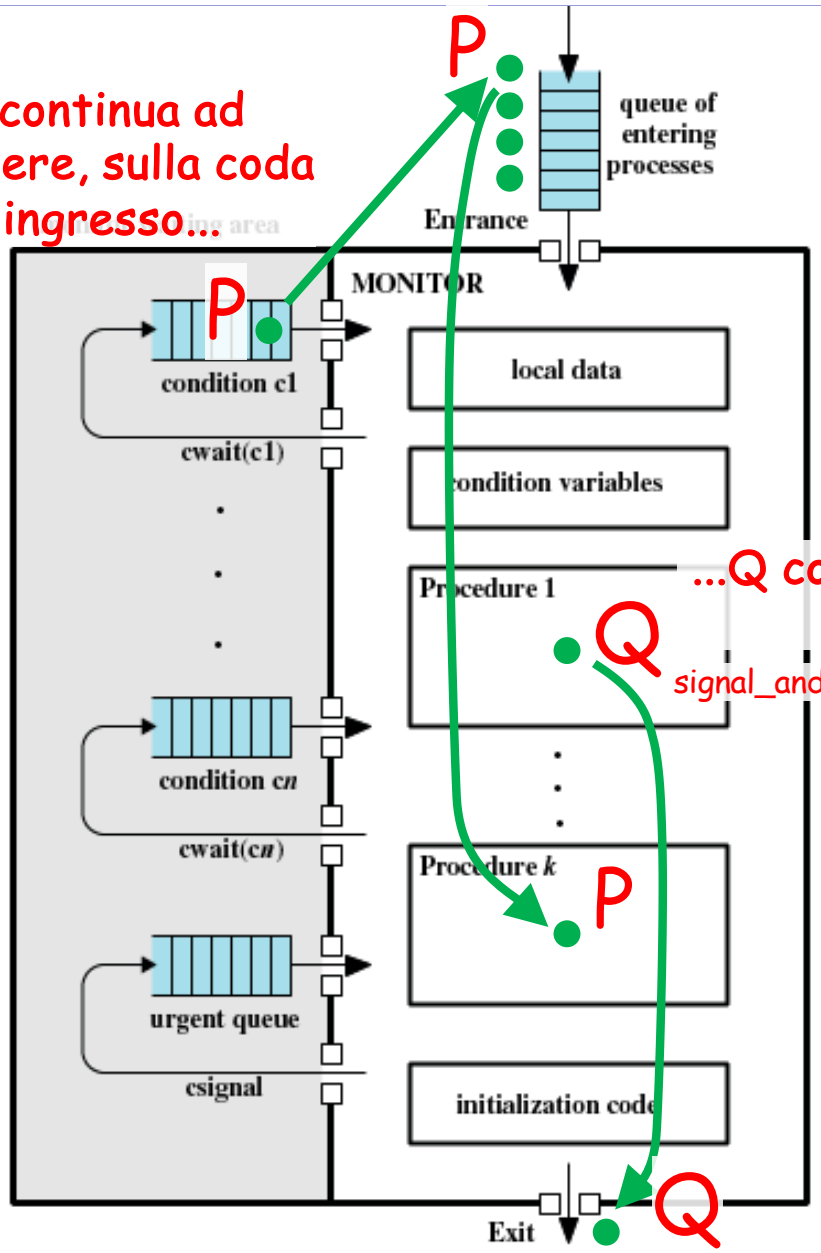
Seconda soluzione: signal and continue

- *Signal_and_continue* (detto anche *wait and notify*):
  - privilegia il processo **segnalante** rispetto al **segnalato**
  - il processo Q segnalante prosegue la sua esecuzione, mantenendo l'accesso esclusivo al monitor



Q prosegue l'esecuzione dopo aver risvegliato P

Monitor – signal and continue



...P continua ad attendere, sulla coda di ingresso...

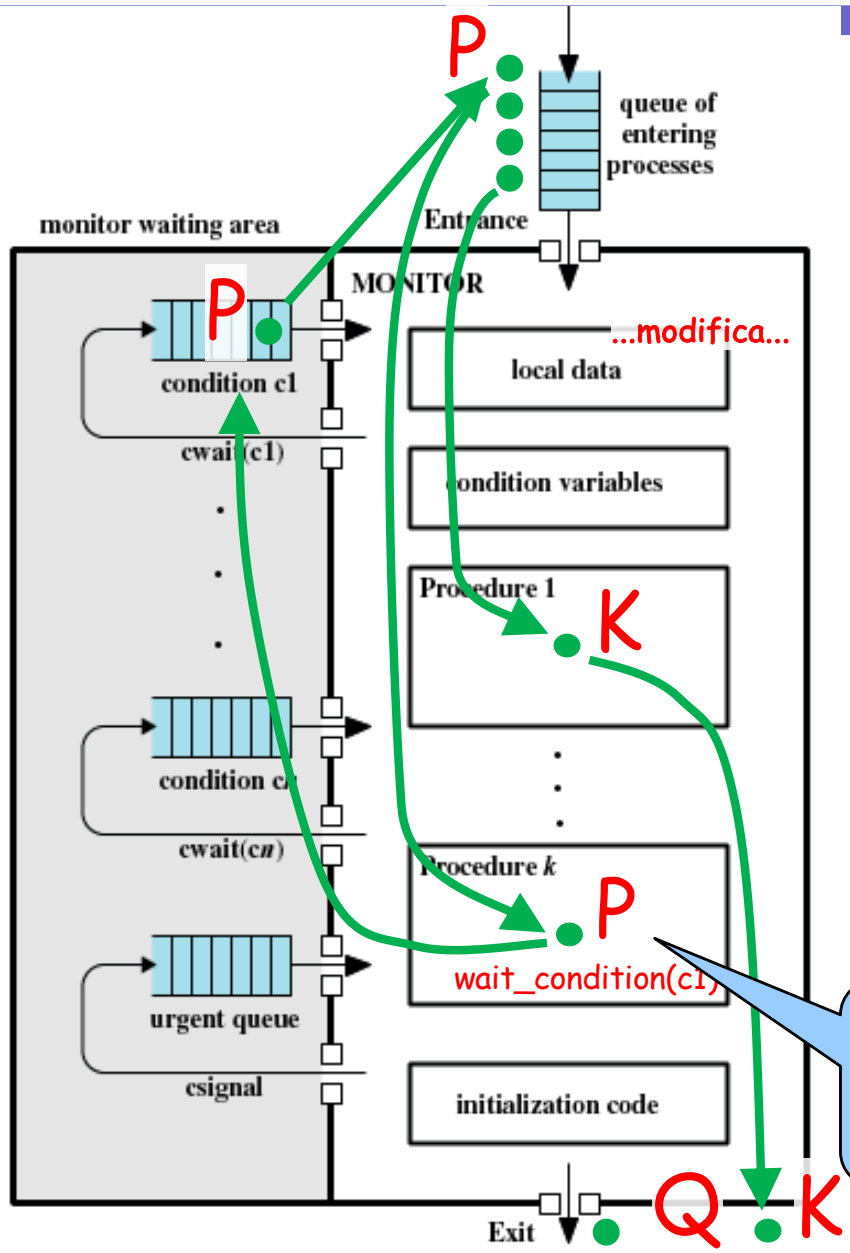
Il processo P segnalato **non viene attivato subito**, ma è trasferito alla coda di **ingresso del monitor**

...Q continua ad eseguire nel monitor...

signal_and_continue(c1)

Il processo P accede al monitor solo dopo l'uscita di Q, e dopo aver conteso l'accesso con altri processi

Monitor – signal and continue



Durante il tempo in cui P è in attesa di entrare, il processo Q (o un altro processo K appena entrato) possono modificare la risorsa

La condizione di sincronizzazione può essere invalidata, costringendo il processo P a una nuova attesa

```
while(condizione non valida) {
    wait_condition(...)
}
```

Signal and continue

- Anche se risvegliato, il processo P **non ha certezza** che la condizione sia verificata
- P deve **controllare nuovamente** la condizione prima di proseguire

Lo schema tipico di uso di wait_cond() è all'interno di un «while»:

```
while (!B) {
    cv.wait_cond();          // È possibile che wait_cond() venga
}                            // chiamata più volte prima di accedere
<...accesso alla risorsa...>
```

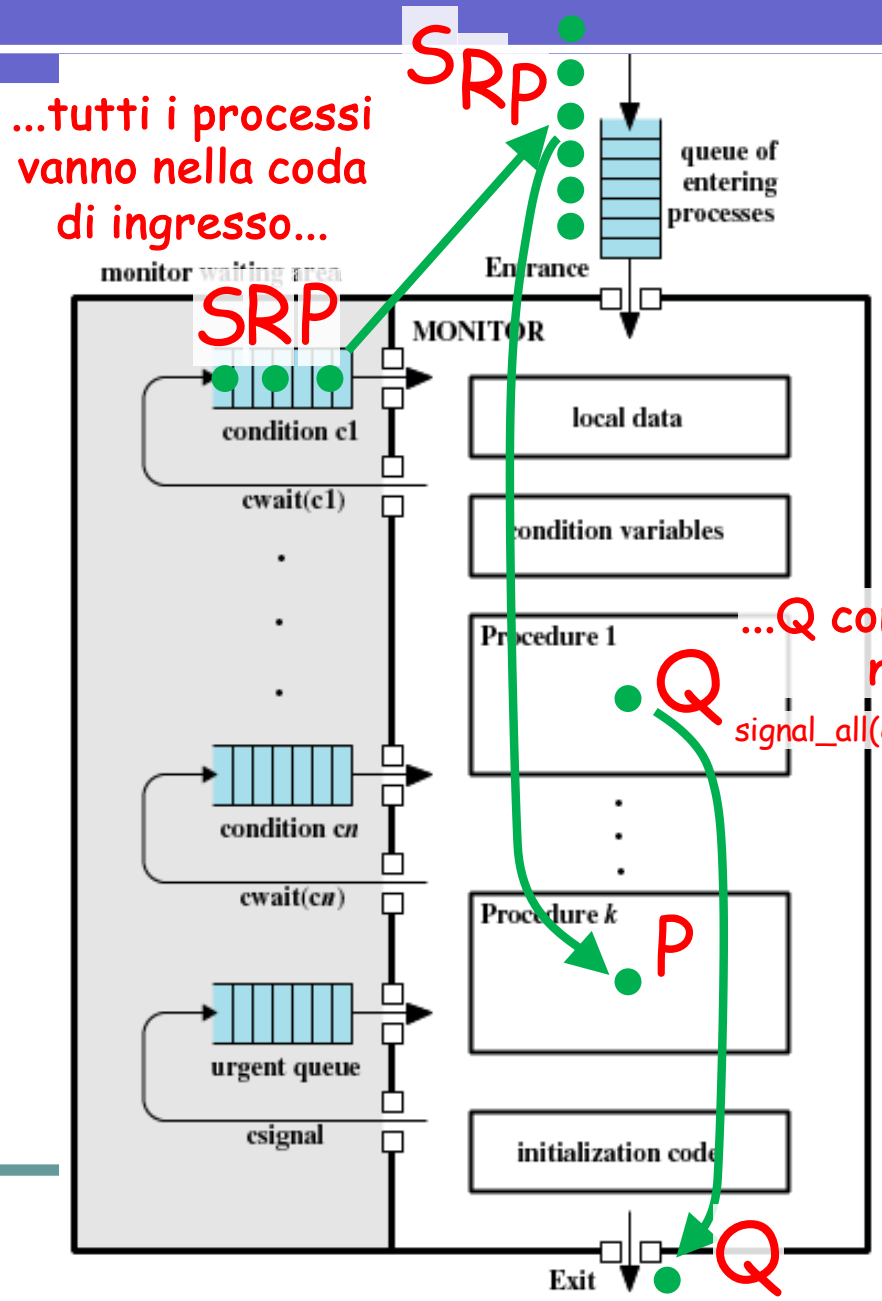
Signal and continue: signal_all

- signal_cond() riattiva al più un solo processo
- È possibile anche risvegliare tutti i processi sospesi sulla variabile condition utilizzando la variante:

signal_all()

- Tutti i processi risvegliati vengono messi nella entry_queue, dalla quale uno alla volta potranno rientrare nel monitor

Monitor – signal all



I processi P, R, S accederanno al monitor uno alla volta.

Ognuno ri-controlla la condizione di sincronizzazione, sospenendosi con *wait_cond()* se necessario

Confronto: signal-and-wait vs signal-and-continue

- La semantica di signal-and-wait richiede che venga chiamata precisamente quando il processo segnalato deve essere svegliato
- La semantica di signal-and-continue (e signal-all) è più robusta
  - il processo segnalante può chiamarla anche quando non è sicuro di se/quali processi risvegliare
  - saranno i processi risvegliati a controllare se possono eseguire, oppure sospendersi

I monitor in UNIX e in C e C++

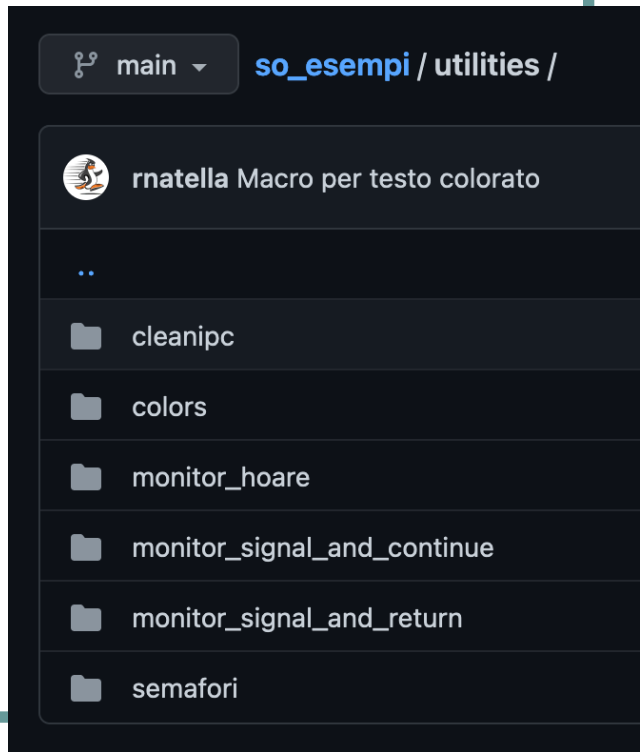
- I sistemi UNIX non forniscono "di serie" delle chiamate di sistema per realizzare i monitor (rimandano ai linguaggi di programmazione)
- Le ultime versioni del C++ forniscono apposite classi, es. std::mutex e std::condition_variable

I monitor in UNIX e in C e C++

- Nei nostri esercizi realizzeremo i monitor in due modi:
  - Nei programmi multi-processo, li implementeremo usando una nostra libreria, basata internamente su semafori e memoria condivisa UNIX
  - Nei programmi multi-thread, useremo la libreria "PThreads" (più avanti nel corso)

Libreria Monitor

- Disponibile su: https://github.com/rnatella/so_esempi
- Sotto-cartella: "**utilities**"



Libreria Monitor

```
struct Monitor {
  /*  Contiene gli ID di almeno un semaforo
   *  (il mutex del monitor), e di ulteriori
   *  shared memory e semafori (per le variabili
   *  condition).
   */
};

void init_monitor (Monitor*, int);
void enter_monitor(Monitor*);
void leave_monitor(Monitor*);
void remove_monitor(Monitor*);
void wait_condition(Monitor*,int);
void signal_condition(Monitor*,int);
```

Libreria Monitor

```
typedef struct {

        int buffer[DIM];
        // ...eventuali altre variabili

        Monitor m;
} MyMonitor;
```

```
MyMonitor * p = shmat(...);

init_monitor(&p->m, 2);
...
remove_monitor(&p->m);
```

l'oggetto-monitor sarà condiviso fra processi

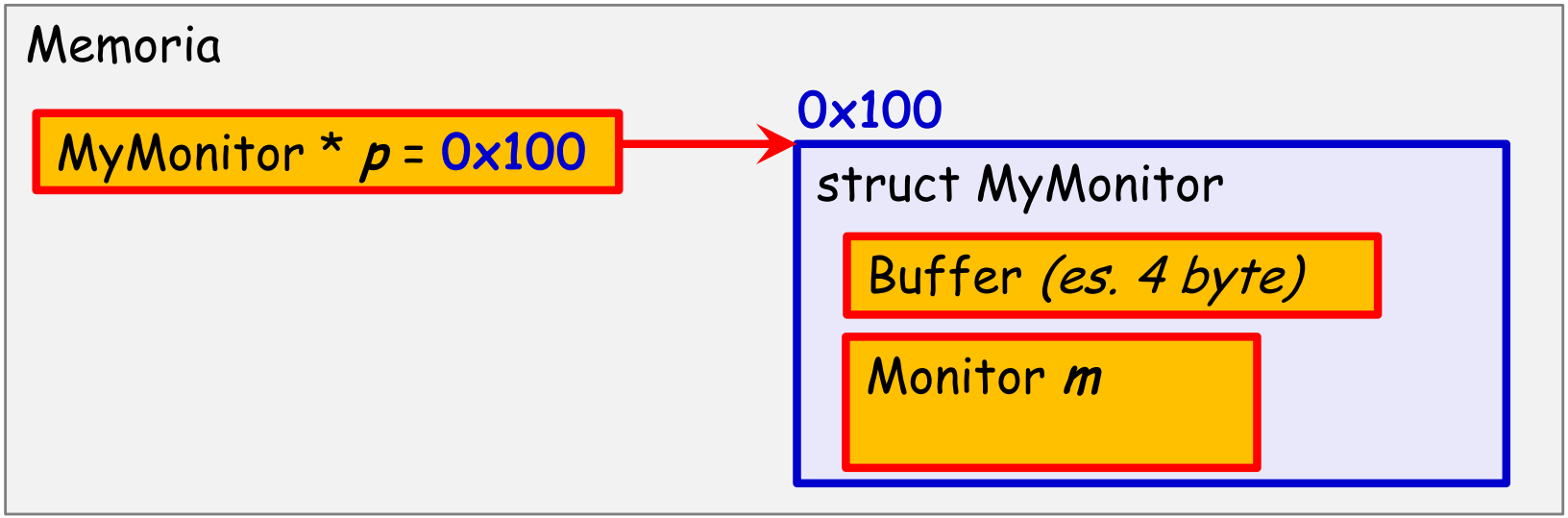
il secondo parametro indica la quantità di variabili condition

Libreria Monitor

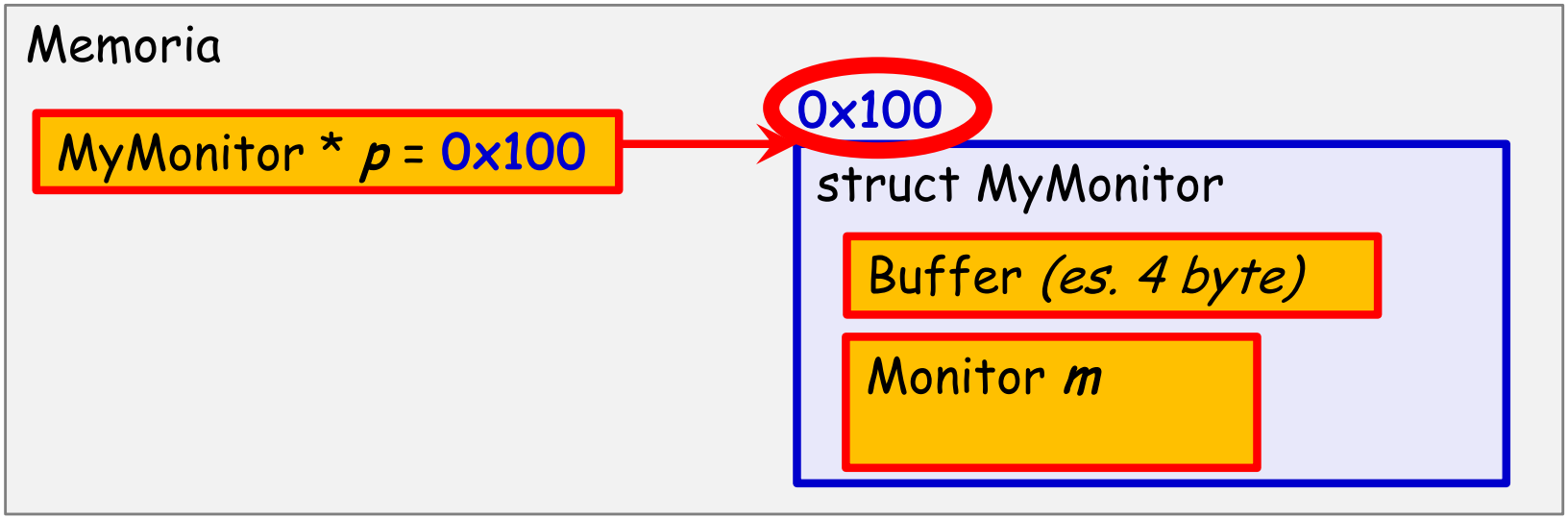
```
#define CV_PRODUTTORI   0
#define CV_CONSUMATORI  1
```

```
metodo(MyMonitor * p, ....) {
    enter_monitor(&p->m);

    if/while(...) {
        wait_condition(&p->m, CV_PRODUTTORI);
    }

    ...
    signal_condition(&p->m, CV_CONSUMATORI);
    leave_monitor(&p->m);
}
```

Libreria Monitor



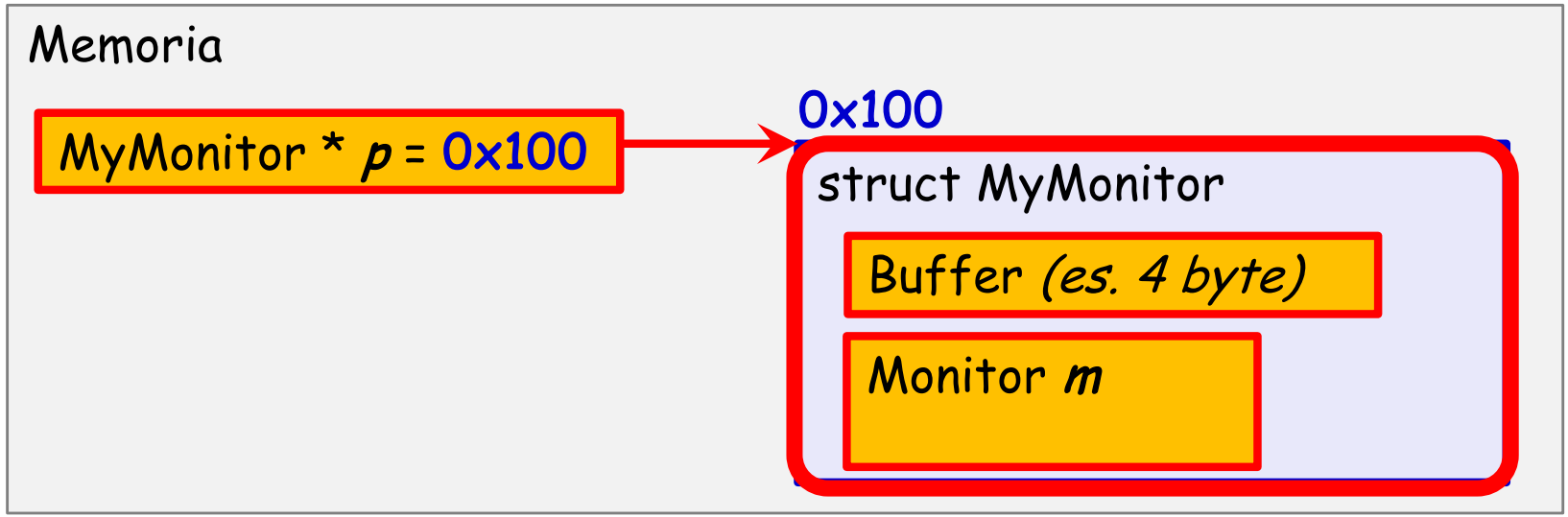
Libreria Monitor



p = 0x100

p è una "variabile-puntatore", contiene l'indirizzo dell'oggetto MyMonitor

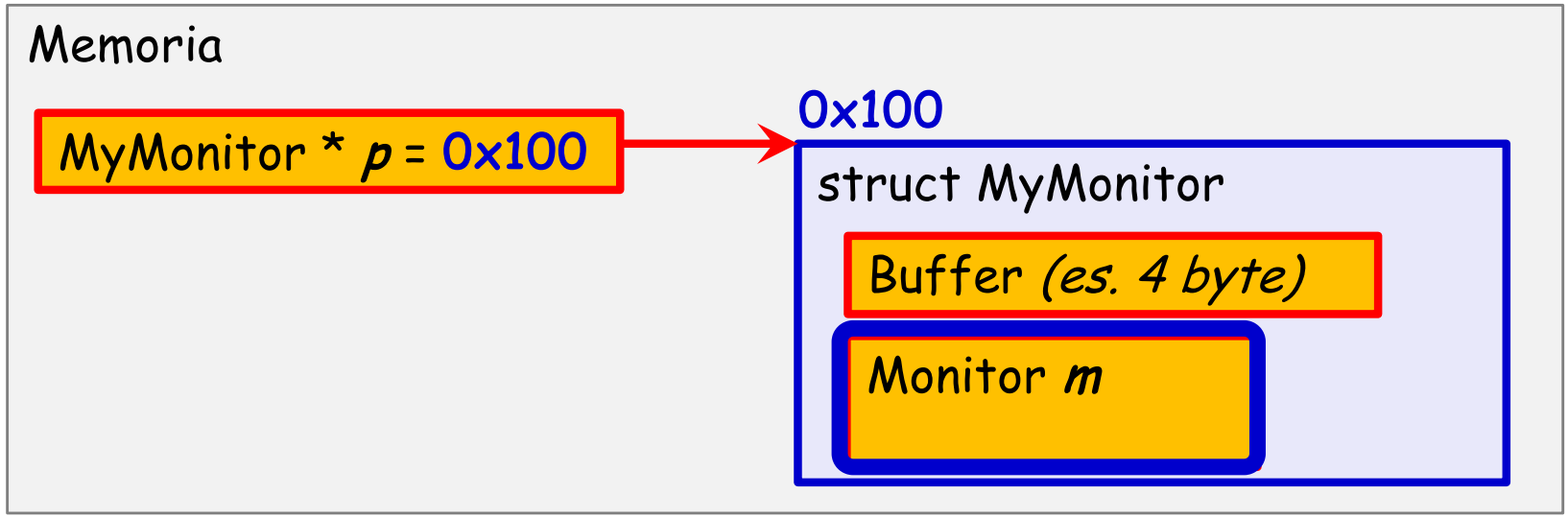
Libreria Monitor



*p
p->... = contenuto di **MyMonitor**

Gli operatori "asterisco" e "freccia" rappresentano
il contenuto dell'oggetto puntato

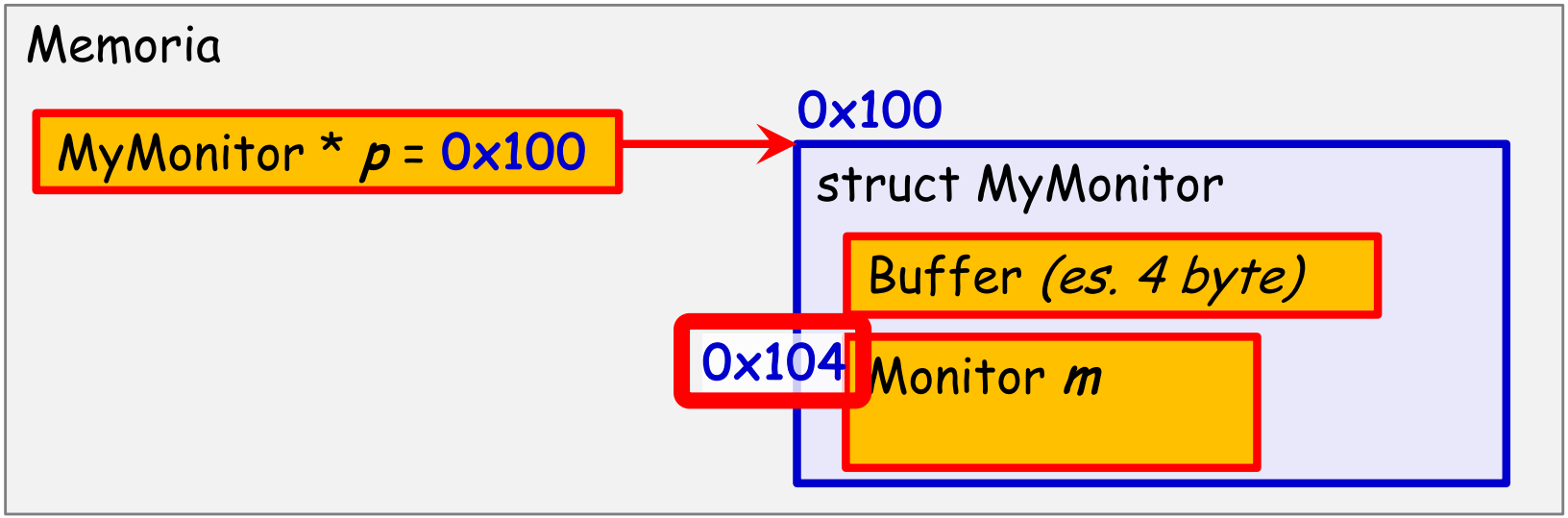
Libreria Monitor



p->m = Monitor m

p->m rappresenta il contenuto della variabile-membro "m"

Libreria Monitor



&(p->m) = indirizzo di Monitor m

= 0x104

&(p->m) rappresenta l'indirizzo della variabile-membro "m"

Produttori-Consumatori su buffer singolo, monitor con semantica signal-and-wait

```
struct buff {
   msg buffer;
   int buffer_pieno=0;
   int buffer_vuoto=1;
   Monitor m;
   // 2 cond: CV_PROD, CV_CONS
}

void Produzione (buff* b, msg mx){
   enter_monitor(&b->m);
   if (b->buffer_pieno==1)
        wait_cond(&b->m, CV_PROD);

   b->buffer = mx;
   b->buffer_pieno = 1;
   b->buffer_vuoto = 0;

   signal_cond(&b->m, CV_CONS);
   leave_monitor(&b->m);
}
...
```

```
 ...
msg Consumo (buff* b) {
   enter_monitor(&b->m);
   msg mx;
   if (b->buffer_vuoto==1)
        wait_cond(&b->m, CV_CONS);

   mx = b->buffer;
   b->buffer_vuoto = 1;
   b->buffer_pieno = 0;

   signal_cond(&b->m, CV_PROD);
   leave_monitor(&b->m);
   return mx;
}
```

Produttori-Consumatori su buffer singolo, monitor con semantica signal-and-wait

```
struct buff {
    msg buffer;
    int buffer_pieno=0;
    int buffer_vuoto=1;
    Monitor m;
    // 2 cond: CV_PROD, CV_CONS
}

void Produzione (buff* b, msg mx){
    enter_monitor(&b->m);
    if (b->buffer_pieno==1)
        wait_cond(&b->m, CV_PROD);

    b->buffer
    b->bu
    b->b

    sign
    leav
}
...
```

```
msg Consumo (buff* b) {
    enter_monitor(&b->m);
    msg mx;
    if (b->buffer_vuoto==1)
        wait_cond(&b->m, CV_CONS);
}
```

NOTA: la scelta delle variabili di stato (buffer_pieno/vuoto, ...), e il numero di variabili condition (CV_PROD, CV_CONS) è a discrezione del programmatore

NOTA: A differenza dei semafori, la condizione di sincronizzazione appare chiaramente nel programma. wait_cond() è un semplice meccanismo di sospensione.

Invece, con i semafori, wait_sem() includeva al suo interno sia il meccanismo di sospensione, sia una politica basata sulla variabile intera interna.

NOTA: utilizza "if" perché si tratta di un monitor signal-and-wait.

In caso di semantica signal-and-continue, sarebbe stato:

while(b->buffer_pieno==1)
wait_cond(...)

Produttori-Consumatori su vettore di buffer circolare, monitor con semantica signal-and-wait

```
struct buff {
   msg buffer[N];
   int contatore=0;
   int testa=0;
   int coda=0;
   Monitor m;
   // 2 cond: CV_PROD, CV_CONS
}

void Produzione (buff* b, msg mx){
   enter_monitor(&b->m);
   if (b->contatore==N)
        wait_cond(&b->m, CV_PROD);

   b->buffer[b->coda] = mx;
   b->coda = (b->coda+1)% N;
   b->contatore ++;

   signal_cond(&b->m, CV_CONS);
   leave_monitor(&b->m);
}
...
```

```
 ...
msg Consumo (buff* b) {
   enter_monitor(&b->m);
   msg mx;
   if (b->contatore==0)
        wait_cond(&b->m, CV_CONS);

   mx = b->buffer[b->testa];
   b->testa =(b->testa+1)% N;
   b->contatore --;

   signal_cond(&b->m, CV_PROD);
   leave_monitor(&b->m);
   return mx;
}
```

NOTA: è importante aggiornare "contatore" insieme a testa/coda (e in generale, ogni volta che si modifica lo stato della risorsa).

Produttori-Consumatori su vettore di buffer circolare, monitor con semantica signal-and-wait

```
struct buff {
   msg buffer[N];
   int contatore=0;
   int testa=0;
   int coda=0;
   Monitor m;
   // 2 cond: CV_PROD, CV_CONS
}

void Produzione (buff* b, msg mx){
   enter_monitor(&b->m);
   if (b->contatore==N)
         wait_cond(&b->m, CV_PROD);

   b->buffer[b->coda] = mx;
   b->coda = (b->coda+1)% N;
   b->contatore ++;

   signal_cond(&b->m, CV_CONS);
   leave_monitor(&b->m);
}
...
```

```
 ...
msg Consumo (buff* b) {
   enter_monitor(&b->m);
   msg mx;
   if (b->contatore==0)
         wait_c    (&b->m, CV_CONS);

   mx = b->
   b->testa
   b->conta

   signal_cond(&b->m, CV_PROD);
```

NOTA: sarebbe potuto essere:

if(b->coda == b->testa)
wait_cond(...)

NOTA: in alternativa, il controllo sullo stato di "pieno" sarebbe potuto essere:

if(b->coda == (b->testa - 1) % DIM)
wait_cond(...)

Produttori-Consumatori su pool di buffer con stato, monitor con semantica signal-and-wait (1/2)

```
struct buff {
  msg buffer[N];

  enum stato_buf { LIBERO,OCCUPATO,INUSO }
  stato_buf stato[N];

  int numero_occupati=0;
  int numero_liberi=N;
  Monitor m;
   // 2 cond: CV_PROD, CV_CONS
}

void Produzione (buff* b, msg mx){

   int i = IniziaProduzione(b);

   // ...operazione lenta...
   buffer[i] = mx;

   FineProduzione(b,i);
}
...
```

```
...
msg Consumo (buff* b) {

   msg mx;

   int i = IniziaConsumo(b);

   // ...operazione lenta...
   mx = buffer[i];

   FineConsumo(b,i);

   return mx;
 }

...
```

Produttori-Consumatori su pool di buffer con stato, monitor con semantica signal-and-wait (1/2)

```
struct buff {
  msg buffer[N];

  enum stato_buf { LIBERO,OCCUPATO,INUSO }
  stato_buf stato[N];

  int numero_occupati=0;
  int numero_liberi=N;
  Monitor m;
   // 2 cond: CV_PROD, CV_CONS
}

void Produzione (buff* b, msg mx){

   int i = IniziaProduzione(b);

   // ...operazione lenta...
   buffer[i] = mx;

   FineProduzione(b,i);
}
...
```

```
...
msg Consumo (buff* b) {

   msg mx;

   int i = IniziaConsumo(b);

   // ...operazione lenta...
   mx = buffer[i];

   FineConsumo(b,i);
```

NOTA: le funzioni Inizia/Fine Produzione/Consumo entrano "temporaneamente" nel monitor, modificano le variabili di stato, ed escono presto dal monitor

NOTA: i processi producono/consumano senza tenere occupato l'oggetto-monitor. Questo approccio consente operazioni in parallelo su buffer distinti

Produttori-Consumatori su pool di buffer con stato, monitor con semantica signal-and-wait (2/2)

```
...
int IniziaProduzione (buff* b){
  enter_monitor(&b->m);
  int i = 0; //indice
  if (b->numero_liberi==0)
    wait_cond(&b->m, CV_PROD);

  while(i<N && b->stato[i]!=LIBERO)
    i++;

  b->stato[i] = INUSO;
  b->numero_liberi--;

  leave_monitor(&b->m);
  return i;
}

void FineProduzione(buff*b,int i){
  enter_monitor(&b->m);
  b->stato[i] = OCCUPATO;
  b->numero_occupati++;
  signal_cond(&b->m, CV_CONS);
  leave_monitor(&b->m);
}
```

```
int IniziaConsumo (buff* b){
   enter_monitor(&b->m);
   int i = 0;
   if (b->numero_occupati==0)
     wait_cond(&b->m, CV_CONS);

   while(i<N && b->stato[i]!=OCCUPATO)
     i++;

   b->stato[i] = INUSO;
   b->numero_occupati--;

   leave_monitor(&b->m);
   return i;
 }

void FineConsumo (buff* b, int i){
   enter_monitor(&b->m);
   b->stato[i] = LIBERO;
   b->numero_liberi++;
   signal_cond(&b->m, CV_PROD);
   leave_monitor(&b->m);
 }
```

Lettori-Scrittori con Monitor (con starvation di entrambi)

```
struct buff {
   msg buffer;
   int num_lett=0;
   int num_scritt=0;
   Monitor m;
   // 2 cond: CV_LETT, CV_SCRITT
}

msg Lettura (buff* b) {

   IniziaLettura(b);

   msg mx = b->buffer;

   FineLettura(b);

   return mx;
}
...
```

NOTA: sia la **Lettura** sia la **Scrittura** sono svolte al di fuori del monitor

- **Lettura** consente accesso in **parallelo** a più lettori.
- **Scrittura**, se c'è già uno scrittore attivo, altri scrittori si sospendono su una variabile condition (in modo da poter gestire la starvation).

```
 ...
void Scrittura (buff* b, msg mx){

   IniziaScrittura(b);

   b->buffer = mx;

   FineScrittura(b);

}
 ...
```

Lettori-Scrittori con Monitor (signal-and-wait, starvation di entrambi)

```
...
int IniziaLettura (buff* b){
  enter_monitor(&b->m);
  if (b->num_scritt > 0)
    wait_cond(&b->m, CV_LETT);

  b->num_lett++;
  signal_cond(&b->m, CV_LETT);

  leave_monitor(&b->m);
}

void FineLettura (buff* b){
  enter_monitor(&b->m);
  b->num_lett--;
  if (b->num_lett==0)
    signal_cond(&b->m,CV_SCRITT);
  leave_monitor(&b->m);
}
```

NOTA: ogni lettore "sveglia" un altro lettore (il 1° lettore riattiva il 2° lettore in attesa; il secondo lettore riattiva il terzo lettore; etc.)

```
int IniziaScrittura (buff* b){
   enter_monitor(&b->m);
   if(b->num_lett>0 || b->num_scritt>0)
     wait_cond(&b->m, CV_SCRITT);

   b->num_scritt++;
   leave_monitor(&b->m);
 }

 void FineScrittura (buff* b){
   enter_monitor(&b->m);
   b->num_scritt--;
   if (queue_cond(&b->m,CV_SCRITT)>0)
     signal_cond(&b->m, CV_SCRITT);
   else (queue_cond(&b->m,CV_LETT)>0)
     signal_cond(&b->m, CV_LETT);
   leave_monitor(&b->m);
 }
```

NOTA: lo scrittore tenta prima di riattivare eventuali altri scrittori in attesa se presenti (per bilanciare la starvation); altrimenti, riattiva i lettori.

Lettori-Scrittori con Monitor (signal-and-continue + signal_all, starvation di entrambi)

```
...
int IniziaLettura (buff* b){
  enter_monitor(&b->m);
  while(b->num_scritt > 0)
    wait_cond(&b->m, CV_LETT);

  b->num_lett++;
  leave_monitor(&b->m);
}
```

NOTA: non c'è più bisogno della *signal_cond(CV_LETT)* grazie alla *signal_all* in FineScrittura.

```
void FineLettura (buff* b){
  enter_monitor(&b->m);
  b->num_lett--;
  if (b->num_lett==0)
    signal_cond(&b->m,CV_SCRITT);
  leave_monitor(&b->m);
}
```

```
int IniziaScrittura (buff* b){
  enter_monitor(&b->m);
  while(b->num_lett>0 ||
                b->num_scritt>0)
    wait_cond(&b->m, CV_SCRITT);

  b->num_scritt++;
  leave_monitor(&b->m);
}

void FineScrittura (buff* b){
  enter_monitor(&b->m);
  b->num_scritt--;
  if (queue_cond(&b->m,CV_SCRITT)>0)
    signal_cond(&b->m, CV_SCRITT);
  else (queue_cond(&b->m,CV_LETT)>0)
    signal_all(&b->m, CV_LETT);
  leave_monitor(&b->m);
}
```

NOTA: con signal_all, lo scrittore riattiva tutti i lettori in attesa (viene chiamata se non vi siano altri scrittori in attesa)