

Il monitor

Il costrutto monitor [Hoare 74]

Definizione: Costrutto sintattico che associa un insieme di operazioni (*entry*, o *public*) ad una struttura dati comune a più processi, tale che:

- Le operazioni **entry** siano le sole operazioni **permesse** su quella struttura.
- Le operazioni **entry** siano **mutuamente esclusive**: un solo processo per volta può essere attivo nel monitor.

Struttura del Monitor

(pseudocodice)

```
monitor tipo_risorsa {  
    <dichiarazioni variabili locali>;  
    <inizializzazione variabili locali>;  
  
    entry void op1 ( ) {  
        <corpo della operazione op1>;  
    }  
    ...  
    entry void opn ( ) {  
        <corpo della operazione opn>;  
    }  
  
    <eventuali operazioni non entry>  
}
```

- Le variabili locali sono accessibili solo all'interno del monitor.
 - Le operazioni entry (o public) sono le sole operazioni che possono essere utilizzate dai processi per accedere alle variabili locali. L'accesso avviene in modo mutuamente esclusivo.
 - Le variabili locali mantengono il loro valore tra successive esecuzioni delle operazioni del monitor (variabili permanenti).
 - Le operazioni non dichiarate entry non sono accessibili dall'esterno. Sono invocabili solo all'interno del monitor (dalle funzioni entry e da quelle non entry).
-

Esempio (*pseudocodice*)

```
tipo_risorsa ris;
```

- crea una istanza del monitor, cioè una struttura dati organizzata secondo quanto indicato nella dichiarazione dei dati locali.

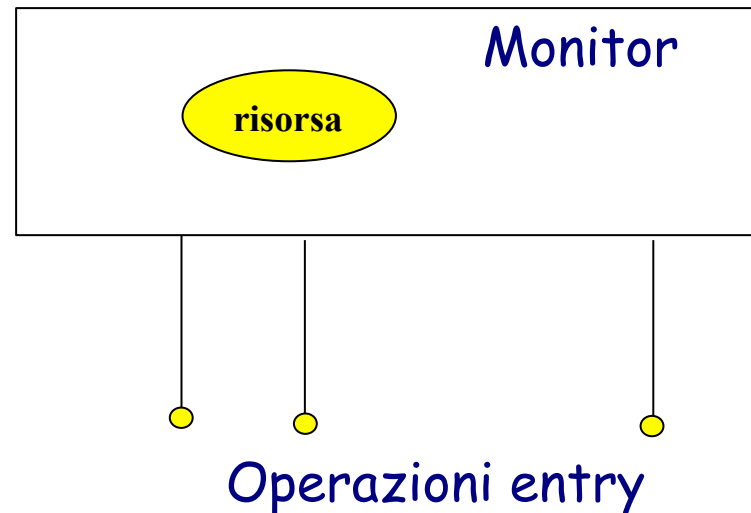
```
ris.opi(...);
```

- chiamata di una generica operazione dell'oggetto ris.

Uso del monitor

Solitamente, al monitor è associata una risorsa:

Scopo del monitor è **controllare l'accesso alla risorsa** da parte processi concorrenti, in accordo a determinate politiche. Le variabili locali definiscono lo stato della risorsa associata al monitor.



1 solo processo alla volta può "entrare" (i.e. eseguire una entry) nel monitor.

P1

P2

Pn

L'accesso alla risorsa avviene tramite il monitor, che garantisce **due livelli di sincronizzazione**:

1. Il **primo** garantisce che un solo processo alla volta possa aver accesso al monitor. Ciò è ottenuto garantendo che le operazioni *entry* siano eseguite in **mutua esclusione** (eventuale sospensione dei processi nella coda *entry queue*).

2. Il **secondo** controlla l'**ordine** con il quale i processi hanno accesso alla risorsa. La procedura chiamata verifica il soddisfacimento di una condizione logica (**condizione di sincronizzazione**) che assicura l'ordinamento. Se la condizione logica non è soddisfatta, il processo viene posto in **attesa** ed il monitor viene liberato.

Monitor: sincronizzazione dei processi

Esempio: allocazione di una risorsa con priorità

```
monitor Risorsa()
```

```
{ boolean risorsa_libera=true;
```

```
  int turno=..;
```

```
  ...
```

```
  entry void acquisizione(int id)
```

```
  { if (turno!=id)
```

```
      <il processo esce dal monitor e aspetta >
```

```
      risorsa_libera=false;
```

```
      <attribuzione nuovo valore a turno>
```

```
  }
```

```
  entry void rilascio(int id)
```

```
  { risorsa_libera=true;
```

```
      <eventuale attribuzione nuovo valore a turno>
```

```
      <risveglia il più prioritario tra i proc. in attesa>
```

```
  }
```

```
}
```


Monitor: sincronizzazione dei processi

- Il **primo livello** di sincronizzazione (mutua esclusione) viene realizzato direttamente dal linguaggio: ogni primitiva **entry** è sempre mutuamente esclusiva.
- Il **secondo livello** di sincronizzazione viene realizzato dal programmatore in base alle politiche di accesso date, sfruttando un nuovo strumento di sincronizzazione: la **variabile condizione** (**condition**):
 - L'accesso alla risorsa controllata dal monitor (da parte di un processo che esegue una entry) è vincolato al soddisfacimento di una **condizione di sincronizzazione**;
 - Nel caso in cui la condizione di sincronizzazione non sia verificata, il processo **si sospende** liberando il monitor; la sospensione del processo avviene tramite una **variabile condizione**.

Variabili tipo condizione

Una variabile **condizione** rappresenta una coda nella quale i processi possono sospendersi (se la condizione di sincronizzazione non è verificata).

Definizione di una variabile **cond** di tipo condizione:

```
condition cond;
```

Operazioni sulle variabili condition:

- E' possibile applicare ad ogni variabile condizione due operazioni:

```
wait(cond) ;
```

```
signal(cond) ;
```

Operazioni sulle variabili condizione

wait:

- L'esecuzione dell'operazione **wait(cond)** sospende il processo, introducendolo nella coda individuata dalla variabile **cond**, e il monitor viene liberato. Al risveglio, il processo riacquisisce l'accesso mutualmente esclusivo al monitor e riprende l'esecuzione.

signal:

- L'esecuzione dell'operazione **signal(cond)** *riattiva* un processo in attesa nella coda individuata dalla variabile **cond**; se non vi sono processi in coda, non produce effetti.

Monitor: uso di wait e signal

Esempio:

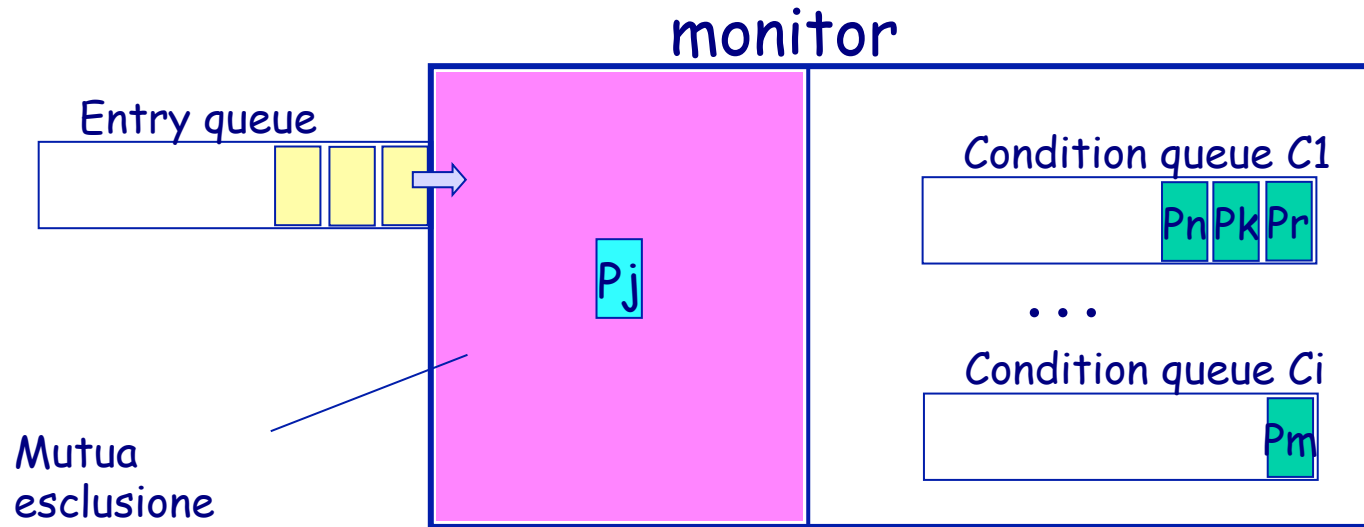
```
monitor Risorsa()
{  boolean risorsa_libera=true;
   condition C;
   int turno=..;
   ...
   entry void acquisizione(int id)
   {   while (turno!=id)
        C.wait();
        risorsa_libera=false;
        <attribuzione nuovo valore a turno>
    }
   entry void rilascio(int id)
   {   risorsa_libera=true;
        <eventuale attribuzione nuovo valore a turno>
        C.signal();
    }}
}
```

Accesso al monitor: code

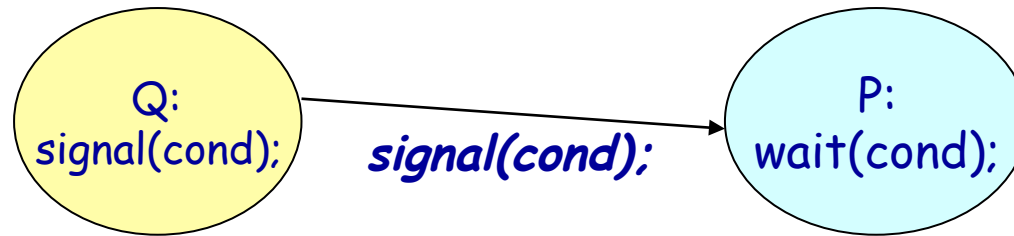
Il controllo nell'accesso al monitor viene esercitato tramite la sospensione dei processi in alcune code:

Primo livello (mutua esclusione): se un processo che vuole accedere al monitor (tramite un'operazione entry) lo trova occupato, esso viene sospeso nella entry queue

Secondo livello: se la condizione di sincronizzazione di un processo che esegue nel monitor (tramite un'operazione entry) non è soddisfatta, esso viene sospeso nella condition associata alla condizione di sincronizzazione (condition queue).



Semantiche dell'operazione signal



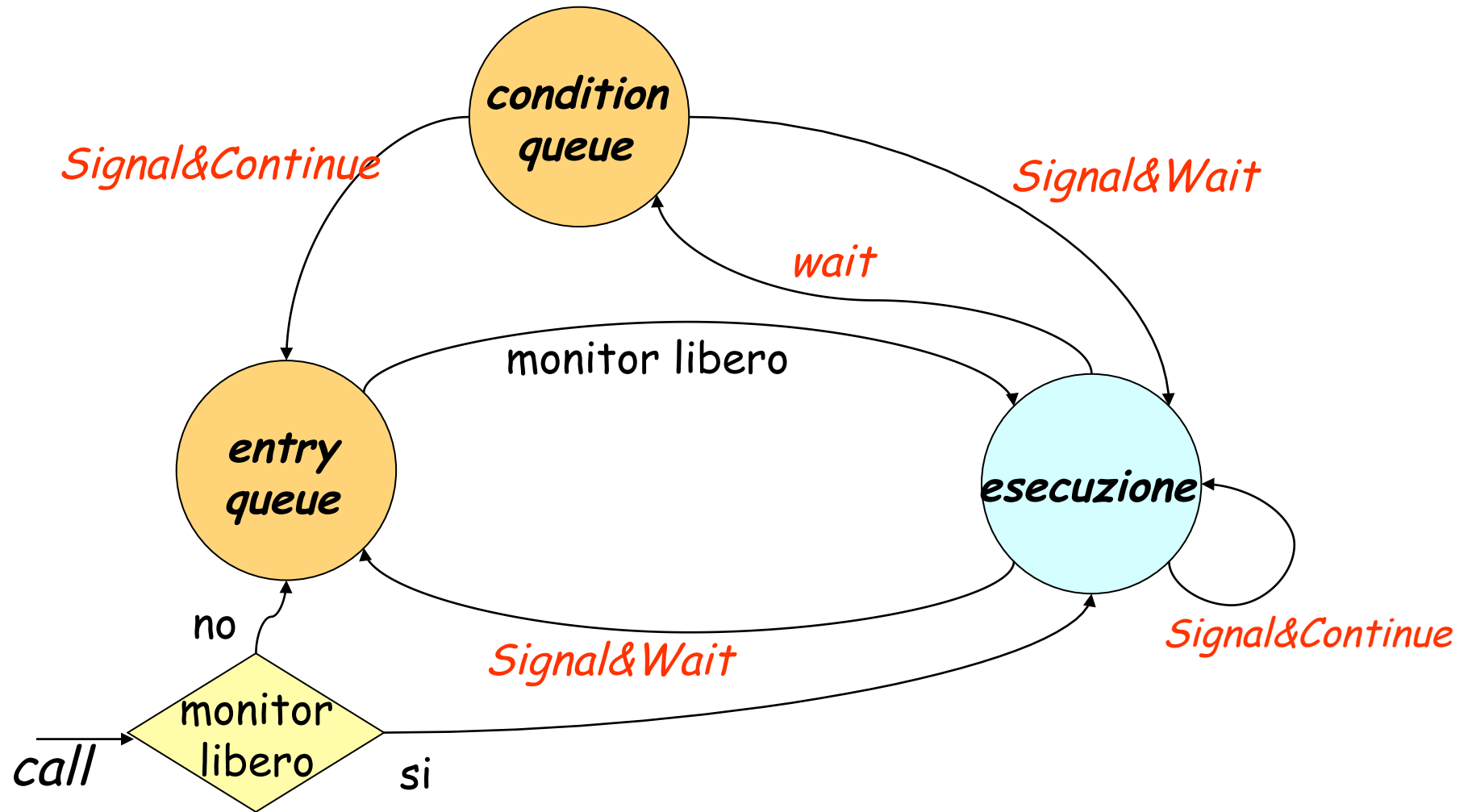
Come conseguenza della `signal` entrambi i processi, quello segnalante Q e quello segnalato P, **possono concettualmente proseguire la loro esecuzione.**

Possibili strategie:

`signal_ and_ wait.` P riprende immediatamente l'esecuzione ed il processo Q viene sospeso.

`signal_ and_ continue.` Q prosegue la sua esecuzione mantenendo l'accesso esclusivo al monitor, dopo aver *risvegliato* il processo .

Semantiche della signal



Signal_and_wait

- Q si sospende nella coda dei processi che attendono di usare il monitor (*entry queue*).
- Il primo processo ad operare nel monitor dopo la signal è certamente P:
 - » Non è possibile che Q o altri processi possano modificare la condizione di sincronizzazione prima che P termini l'esecuzione della operazione entry.

signal_and_continue

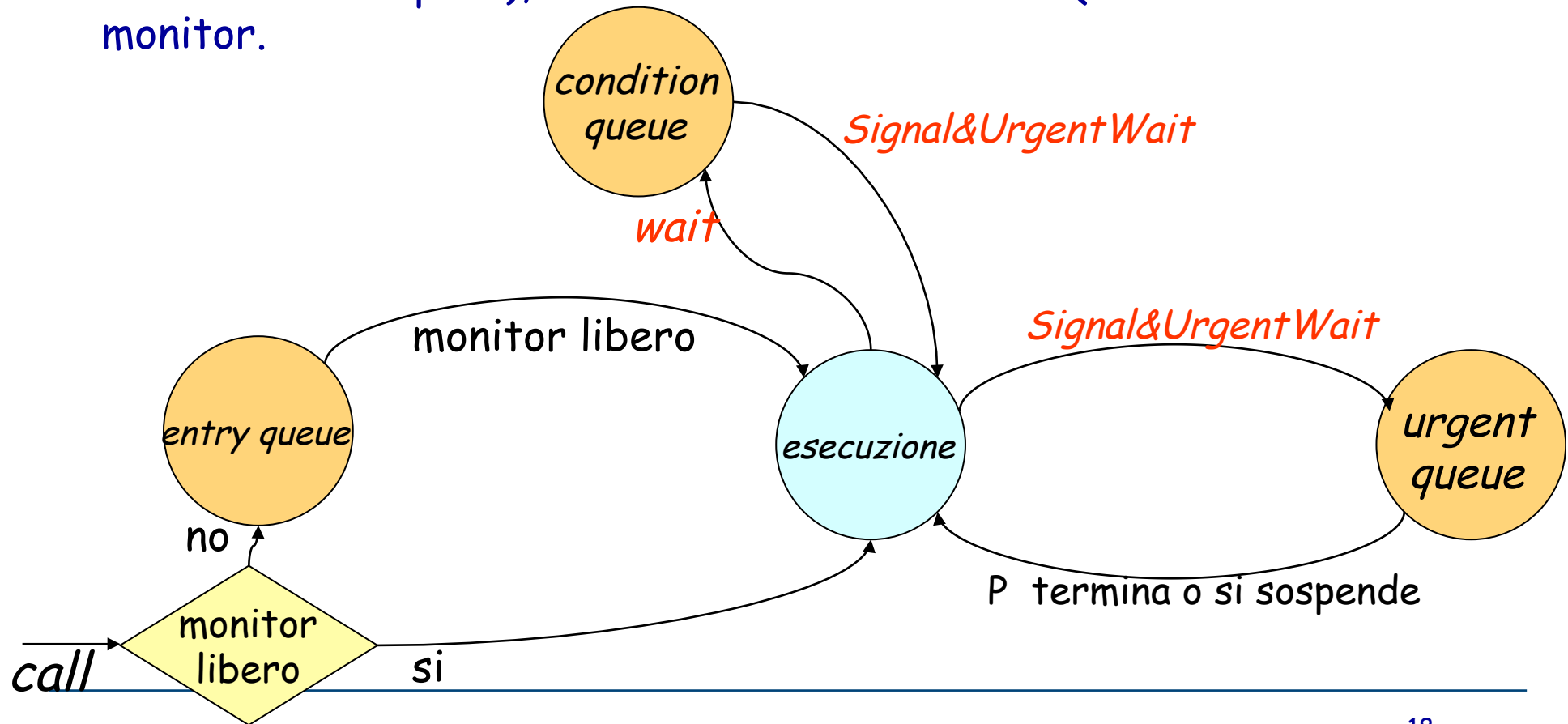
- Il processo segnalato P viene trasferito dalla coda associata alla variabile condizione alla entry_queue e potrà rientrare nel monitor una volta che Q l'abbia rilasciato.
- Poiché altri processi possono entrare nel monitor prima di P, questi potrebbero modificare la condizione di sincronizzazione (lo stesso potrebbe fare Q).
- E' pertanto necessario che quando P rientra nel monitor ritesti la condizione:

```
while(!B) wait (cond);  
<accesso alla risorsa>
```

Signal_and_urgent_wait

signal_and_urgent_wait. E' una variante della signal_and_wait:

Q ha la priorità rispetto agli altri processi che aspettano di entrare nel monitor. Viene quindi sospeso in una coda interna al monitor (*urgent queue*). Quando P ha terminato la sua esecuzione (o si è nuovamente sospeso), trasferisce il controllo a Q senza liberare il monitor.



- Un caso particolare della `signal _and_urgent_wait` (e della `signal_and_wait`) si ha quando essa corrisponde ad una istruzione `return`: **`signal_and_return`**.
- Il processo completa cioè la sua operazione con il risveglio del processo segnalato. Cede ad esso il controllo del monitor senza rilasciare la mutua esclusione.

- E' possibile anche **risvegliare tutti i processi sospesi** sulla variabile condizione utilizzando la :

signal_all

che è una variante della signal_and_continue.

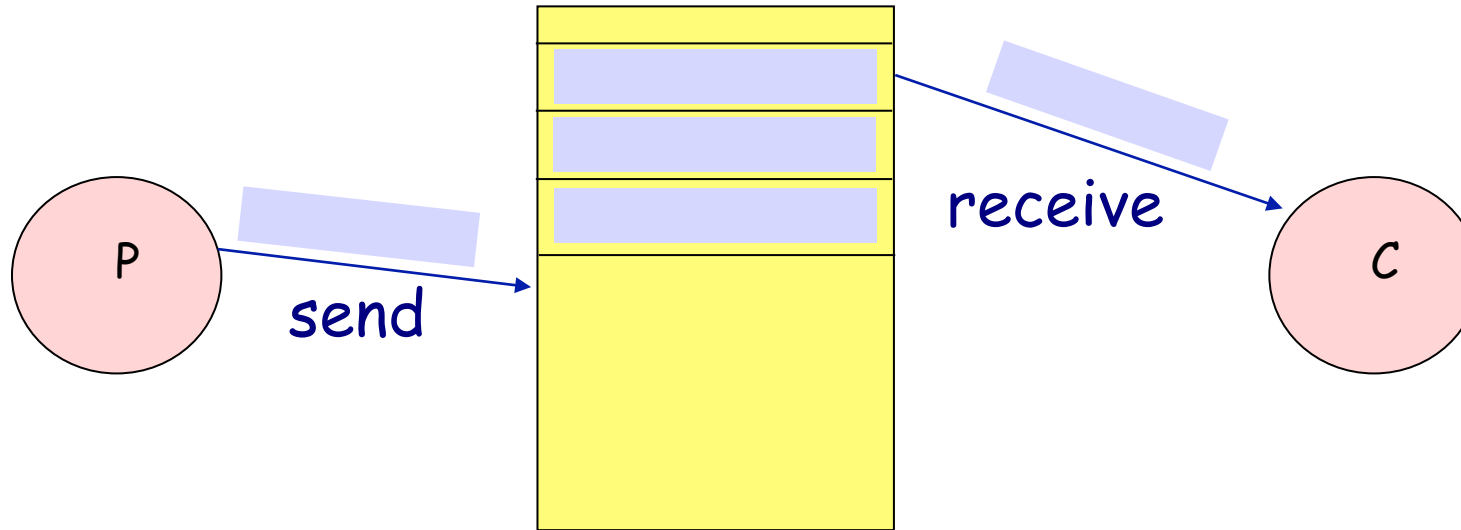
- Tutti i processi risvegliati vengono messi nella entry_queue dalla quale, uno alla volta potranno rientrare nel monitor.

Esempio: monitor come *gestore* di risorse (mailbox)

Utilizziamo il monitor per risolvere il problema della comunicazione tra processi mediante un buffer di dimensione N ("*produttori e consumatori*");

- la struttura dati che rappresenta il buffer fa parte delle variabili locali al monitor e quindi le operazioni *Send* e *Receive* possono accedere solo in modo **mutuamente esclusivo** a tale struttura.
- il monitor rappresenta il buffer dei messaggi (gestito in modo circolare)
- i processi Produttori (o Consumatori) inseriranno (o preleveranno) i messaggi mediante le funzioni entry *Send* (o *Receive*) definite nel monitor.

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

[*HP: semantica signal&wait*]

```

monitor buffer_circolare{
    messaggio buffer[N];
    int contatore=0; int testa=0; int coda=0;
    condition non_pieno;
    condition non_vuoto;

    /* procedure e funzioni entry: */
    entry void send(messaggio m){ /*proc. entry -> mutua esclusione*/
        if (contatore==N) non_pieno.wait;
        buffer[coda]=m;
        coda=(coda + 1)%N;
        ++contatore;
        non_vuoto.signal;
    }

    entry messaggio receive() { /*proc. entry -> mutua esclusione*/
        messaggio m;
        if (contatore == 0) non_vuoto.wait;
        m=buffer[testa];
        testa=(testa + 1)%N;
        --contatore;
        non_pieno.signal;
        return m;}
}/* fine monitor */

```

Se la semantica fosse **signal&continue ??**

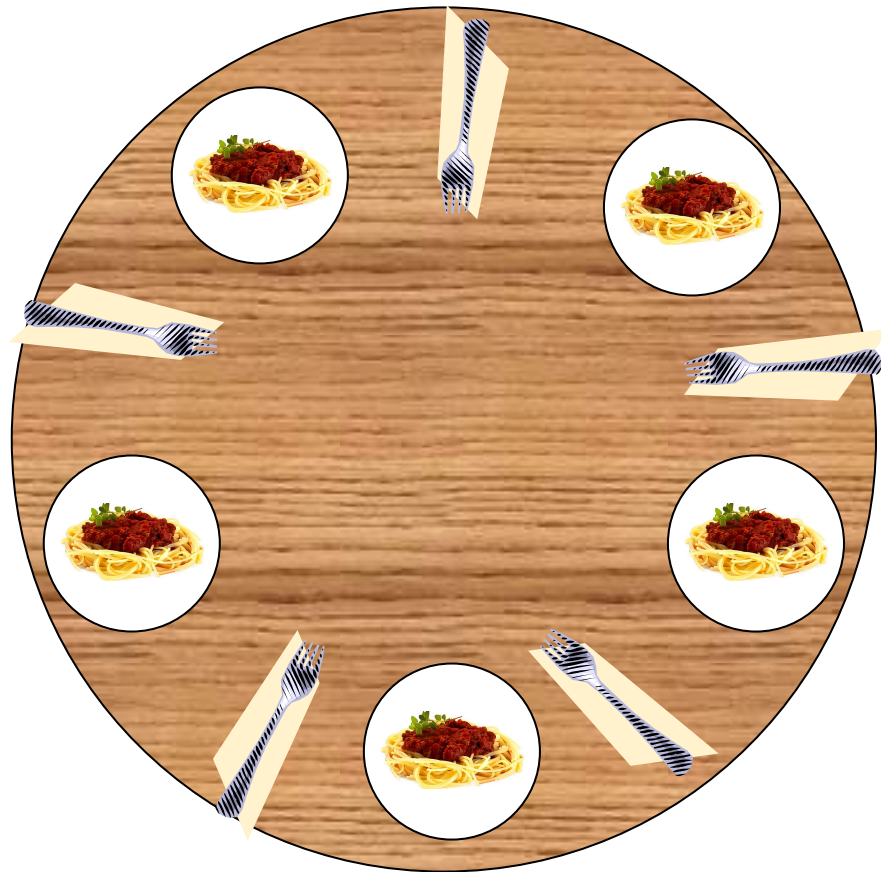
Esempio di uso del costrutto monitor

I filosofi a cena
(E. Dijkstra, 1965)

Il problema

- 5 filosofi sono seduti attorno a un tavolo circolare; ogni filosofo ha un piatto di spaghetti tanto scivolosi che necessitano di 2 forchette per poter essere mangiati; sul tavolo vi sono in totale 5 forchette.
- Ogni filosofo ha un comportamento ripetitivo, che alterna due fasi:
 - una fase in cui **pensa**,
 - una fase in cui **mangia**.

Rappresentando ogni filosofo con un thread, realizzare una politica di sincronizzazione che eviti situazioni di deadlock.



Osservazioni

- i filosofi non possono mangiare tutti insieme: ci sono solo 5 forchette, mentre ne servirebbero 10;
- 2 filosofi vicini non possono mangiare contemporaneamente perche` condividono una forchetta e pertanto quando uno mangia, l'altro e` costretto ad attendere

Soluzione n.1

Quando un filosofo ha fame:

1. prende la forchetta a sinistra del piatto
2. poi prende quella che a destra del suo piatto
3. mangia per un po'
4. poi mette sul tavolo le due forchette.

→ **Possibilita' di deadlock:** se tutti i filosofi afferrassero contemporaneamente la forchetta di sinistra, tutti rimarrebbero in attesa di un evento che non si potra' mai verificare.

Soluzione n.2

Ogni filosofo verifica se entrambe le forchette sono disponibili:

- ❑ in caso affermativo, acquisisce le due forchette (in modo atomico);
 - ❑ in caso negativo, aspetta.
- in questo modo non si può verificare deadlock (non c'è possesso e attesa)

Realizzazione soluzione 2

Quali processo?

- ▣ filosofo

Risorsa condivisa?

la tavola apparecchiata

-> definiamo la classe **tavola**, che
rappresenta il monitor allocatore delle
forchette

Struttura Filosofo_i

```
tavola m;// istanza del monitor
```

```
process filosofo {  
    while(true)  
    { m.prendiForchette(i) ;  
      <mangia...>  
      m.rilasciaForchette(i) ;  
      <pensa...>  
    }  
}
```

Monitor

```
monitor tavola
{
    int forchette[5] = {2,2,2,2,2};
    //le forchette disponibili per ogni filosofo i
    //inizialmente sono 2

    condition codaF[5]; //1 coda per ogni filosofo i

    // metodi entry :
    entry void prendiForchette(int i) {...}
    entry void rilasciaForchette(int i) {...}

    // metodi privati:
    int destra(int i) {...}
    int sinistra(int i) {...}
}
```


Metodi entry

```
entry void prendiForchette(int i)
{
    while (forchette[i] != 2)
        wait(codaF[i]);

    forchette[sinistra(i)]--;
    forchette[destra(i)]--;

}
```

```
entry void rilasciaForchette(int i)
{
    forchette[sinistra(i)]++;
    forchette[destra(i)]++;
    if (forchette[sinistra(i)]==2)
        signal(codaF[sinistra(i)]);
    if (forchette[destra(i)]==2)
        signal(codaF[destra(i)]);
}
```

Metodi privati

```
int destra(int i)
{ int ret;
  if (i==0)
    ret=NF-1;
  else ret=i-1;
  return ret;
}
```

```
int sinistra(int i)
{ int ret;
  ret=(i+1)%NF;
  return ret;
}
```

Ulteriori operazioni sulle variabili condizione

Sospensione con indicazione della priorità:

`wait(cond, p);`

- i processi sono accodati rispettando il valore (crescente o decrescente) di p e vengono risvegliati nello stesso ordine.

Verifica dello stato della coda:

`queue(cond);`

- fornisce il valore vero se esistono processi sospesi nella coda associata a cond, true altrimenti.

Esempio: allocazione di risorse in uso esclusivo

Si vuole che una risorsa venga assegnata a quello tra tutti i processi sospesi che la userà per il periodo di tempo inferiore:

```
monitor allocatore
{
    boolean occupato = false;
    condition libero;

    entry void Richiesta(int tempo)
    {
        if (occupato) libero.wait(tempo);
        occupato = true;
    }
    entry void Rilascio()
    {
        occupato = false;
        libero.signal;
    }
}
```

I processi sono inseriti nella coda secondo l'ordine crescente di p e quindi il primo processo risvegliato è quello che richiede meno tempo.
