

Monitor [Hoare 74]

- Costrutto sintattico che associa un insieme di procedure/funzioni (**entry**) ad una struttura dati comune a più processi.
- Il compilatore può verificare che esse siano *le sole operazioni* permesse su quella struttura .
- Le procedure sono *mutuamente esclusive*: un solo processo per volta può essere attivo nel monitor.

```
monitor <nome_monitor>
{ <dichiarazione delle variabili locali>;
<inizializzazione delle variabili locali>;
/* definizione delle funzioni e procedure entry:
*/
public void op1() /* procedura "entry" */
{ <corpo della funzione op1>; }
...
public void opN() { /* procedura "entry" */
<corpo della funzione opn>; }

/* funzioni&procedure interne al monitor:
void Pr1(..){...}
void Pr2(..){...}
...
}
```

Procedure/funzioni entry: sono *le sole operazioni* che possono essere utilizzate dai processi per accedere alle *variabili locali al monitor*.

Variabili locali: mantengono il loro valore tra successive esecuzioni delle procedure del monitor (*variabili permanenti*); sono accessibili **solo entro il monitor** (mediante funzioni/procedure entry e non entry).

Procedure interne (non entry): non sono invocabili dall'esterno. Sono usabili **solo dalle procedure/funzioni del monitor (entry, o interne)**.

Inizializzazione delle variabili locali: il codice per l'inizializzazione delle variabili locali viene eseguito *una sola volta* prima dell'esecuzione di qualunque procedura.

Uso del monitor

Il monitor puo' essere utilizzato per controllare gli accessi a una **risorsa condivisa** da parte di piu' processi:

- lo **stato della risorsa** viene tipicamente rappresentato dai valori di variabili locali
- i processi possono aggiornare lo stato della risorsa mediante le **procedure entry**

```
monitor Tipo_Risorsa
{ <dichiarazione delle variabili locali>;
<inizializzazione delle variabili locali>;
<definizione funzioni/procedure entry*/
<def. funzioni&procedure interne */
}
```

Uso del monitor

```
Typo_risorsa ris; /* ris e` un oggetto  
di tipo monitor*/
```

- crea un particolare oggetto *monitor*, cioè una struttura dati organizzata secondo quanto indicato nella dichiarazione dei dati locali.
- La chiamata di una generica operazione *opi* dell'oggetto *ris* ha quindi la forma:

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

Uso del monitor

• Scopo del monitor è *controllare l'assegnazione* di una risorsa tra processi concorrenti in accordo a *determinate politiche di gestione*.

- L'assegnazione avviene secondo **due livelli di controllo**:

1. Il primo garantisce che *un solo processo alla volta* possa aver accesso alle variabili comuni del monitor. Ciò è ottenuto automaticamente, poiché le procedure entry sono eseguite in modo *mutuamente esclusivo*. (eventuale *sospensione dei processi*).

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* che assicura l'ordinamento degli accessi (eventuale *sospensione del processo e liberazione del monitor*).

- La *condizione di sincronizzazione* è espressa mediante *variabili locali* al monitor e variabili *proprie* del processo passate come *parametri*.
- La *sospensione del processo*, nel caso in cui la condizione non sia verificata, avviene utilizzando variabili di un nuovo tipo, detto *condition* (*condizione*).

Variabili Condizione

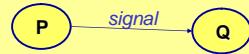
- Una variabile di tipo condizione rappresenta *una coda* di processi sospesi.
- Esistono *tante variabili condizione* quante sono le condizioni per cui un processo può essere ritardato.
- Le procedure del monitor agiscono su tali variabili mediante le 2 operazioni:

cond.wait
cond.signal

Variabili condizione

- **cond.wait:** l'invocazione dell'operazione `cond.wait` da parte di un processo P sospende P e lo introduce nella coda individuata dalla variabile `cond`; prima di sospendersi, P libera il monitor.
- **cond.signal:** l'esecuzione dell'operazione `cond.signal`:
 - se la coda associata a `cond` contiene almeno un processo, rende attivo uno dei processi in attesa nella coda individuata dalla variabile `cond`; alla ripresa dell'esecuzione il processo risvegliato rioccupa automaticamente il monitor.
 - se la coda associata a `cond` è vuota, non provoca alcun effetto.

Signal



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

- **Realizzazione:** Esistono due possibilità di realizzazione della signal su variabili condizione :

Signal and wait: P attende che Q abbandoni il monitor, o che si sospenda per un'altra condizione [Hoare].

Signal and continue: Q attende che P abbandoni il monitor, o che si sospenda per un'altra condizione.

Realizzazione signal

- **Signal and continue** presenta l'inconveniente che quando Q riprende l'esecuzione la condizione logica per la quale stava attendendo potrebbe *non essere più vera* (*se viene modificata da P*).
- **Signal and wait** assicura che P riprenderà la sua esecuzione quando Q avrà completato la sua esecuzione, o si sarà nuovamente sospeso.
- Il monitor verrà liberato *solo quando* non vi saranno al suo interno processi in grado di completare l'esecuzione.
- **Compromesso:** signal eseguita come *ultima istruzione* della procedura -> dopo aver eseguito la `signal`, P abbandona immediatamente il monitor.

Esempio: monitor come gestore di risorse (mailbox)

Utilizziamo il monitor per risolvere il problema dei "**produttori e consumatori**":

- 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.
- 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.

```

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: */
public void send(messaggio m){
    if (contatore==N) non_pieno.wait;
    buffer[coda]=m;
    coda=(coda + 1)%N;
    ++contatore;
    non_vuoto.signal;
}

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

```

Esempio: monitor come allocatore di risorse

- Utilizziamo il monitor per garantire l'accesso esclusivo ad una risorsa comune da parte dei processi.
 - La struttura dati gestita dal monitor rappresenta lo stato (*libero,occupato*) della risorsa.
 - Le operazioni Richiesta e Rilascio del monitor sono utilizzate solo per garantire l'accesso esclusivo alla risorsa da parte dei processi.
 - La mutua esclusione tra le operazioni Richiesta e Rilascio garantisce che lo stato della risorsa venga esaminato in modo mutuamente esclusivo dai processi.
 - Una volta guadagnato l'accesso alla risorsa i singoli processi potranno accedere direttamente ad essa all'esterno del monitor.

```

monitor allocatore
{ boolean occupato = false;
  condition libero;

  public void Richiesta()
  { if (occupato) libero.wait;
    occupato = true;
  }
  public void Rilascio()
  { occupato = false;
    libero.signal;
  }
}

allocatore A; /* istanza del tipo monitor*/

void processo() /*codice di un generico processo */
{
    A.Richiesta;
    <uso della risorsa>;
    A.Rilascio;
}

```

Implementazione del monitor tramite semafori (*signal&wait*)

- Il compilatore assegna ad ogni istanza di monitor:
 - un semaforo **mutex** inizializzato a 1 per la **mutua esclusione** delle procedure entry del monitor;
 - un semaforo **urgent** inizializzato a 0 per effettuare la **preemption** dei processi segnalanti (**signal and wait**);
 - un contatore **urgentcount** inizializzato a 0 per conteggiare in ogni istante i precedenti processi;
 - per ogni variabile **cond** di tipo condition:
 - un semaforo **condsem** inizializzato a 0
 - un contatore **condcount** inizializzato a 0
- per implementare **cond.wait** e **cond.signal**

Realizzazione del monitor

Mutua esclusione delle procedure entry: il compilatore inserisce un prologo ed un epilogo all'inizio ed all'uscita da ogni procedure entry:

- prologo:

```
wait(&mutex);
```

- epilogo:

```
if (urgentcount>0)
    signal(&urgent);
else signal(&mutex);
```

Il compilatore traduce le operazioni **cond.wait** e **cond.signal** nel seguente modo:

```
cond.wait:
condcount++;
if (urgentcount > 0)
    signal(&urgent);
else signal(&mutex)
wait(&condsem);
condcount--;
```

```
cond.signal:
urgentcount++;
if (condcount>0)
{
    signal(&condsem);
    wait(&urgent);
}
urgentcount--;
```

condsem: semaforo associato alla variabile condizione (v.i. = 0)

condcount: contatore associato alla variabile condizione (v.i. = 0)

NB: questa soluzione implementa la politica **signal and wait**

Implementazione signal&continue

- Nel caso di una politica **signal and continue** il semaforo **urgent** ed il contatore **urgentcount** non sono piu` necessari.

```
cond.wait:
condcount++;
signal(&mutex);
wait(&condsem);
condcount--;
```

```
cond.signal:
if (condcount>0)
    signal(&condsem);
else signal(&mutex);
condcount--;
```

La **cond.signal** viene usata come epilogo della procedura entry.

Estensioni al monitor

- In alcuni casi puo` essere utile poter risvegliare i processi sospesi su variabili condizione secondo una priorita` stabilita` arbitrariamente: wait con priorita`
- Al momento della sospensione va specificato un indice di priorita` : **cond.wait(p)**

→ I processi sono inseriti nella coda secondo l'ordine crescente di p: quindi il primo processo risvegliato è quello con il valore di p piu` basso.

- Altre funzioni primitive su variabili di tipo condizione:

cond.queue: operazione che verifica la presenza nella coda **cond** di almeno un processo sospeso

Esempio: **if (cond.queue) ...**

Esempio: allocazione di risorse in uso esclusivo

Si vuole che la 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;

  public void Richiesta(int tempo)
  { if (occupato) libero.wait(tempo);
    occupato = true;
  }
  public 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.

Esempio: lettori e scrittori

Si supponga di voler realizzare la seguente **politica di allocazione della risorsa**:

1. un nuovo lettore non può acquisire la risorsa se c'e' uno scrittore in attesa
2. tutti i lettori sospesi al termine di una scrittura hanno priorità sul successivo scrittore

Soluzione: uso del monitor con le seguenti variabili locali:

- **num-lettori**: il numero di processi lettori attivi sulla risorsa
- **occupato**: una variabile logica che indica se la risorsa è occupata da uno scrittore (occupato=true)
- **ok-lettura, ok-scrittura** : due variabili condizione sulle quali si sospendono rispettivamente i processi lettori e scrittori

```
monitor lettori_scrittori
{ int num_lettori=0,occupato=0;
  condition ok_lettura,ok_scrittura;

  public void inizio_lettura()
  { if (occupato || ok_scrittura.queue)
      ok_lettura.wait;
    num_lettori++;
    ok_lettura.signal;
  }
  public void fine_lettura()
  { num_lettori--;
    if (num_lettori==0)
      ok_scrittura.signal;
  }
  public void inizio_scrittura()
  { if ((num_lettori!=0)|| occupato)
      ok_scrittura.wait;
    occupato=1;
  }
  /* continua...*/
}
```

```
/* ...continua */
public void fine_scrittura()
{ occupato=0;
  if (ok_lettura.queue)
    ok_lettura.signal;
  else ok_scrittura.signal;
}
/* fine monitor */

lettori_scrittori LS; /* istanza del monitor*/

void lettore() /*codice di un generico lettore */
{
  LS.inizio_lettura();
  <lettura>;
  LS.fine_lettura()
}
void scrittore() /*codice di un generico scrittore */
{
  LS.inizio_scrittura();
  <scrittura>;
  LS.fine_scrittura()
}
```

Chiamate innestate a procedure di monitor

- Durante l'esecuzione della procedura **A1** del monitor **A** viene chiamata la procedura **B1** del monitor **B** e durante l'esecuzione di tale procedura il processo viene sospeso sulla variabile **condizione X**.
- Viene rilasciata la mutua esclusione per il monitor **B**, mentre il monitor **A** rimane occupato.
- Se la procedura **B2** di **B**, a cui è demandato il compito di riattivare il processo, viene richiamata solo attraverso la procedura **A2** di **A**, si ha una situazione di **blocco critico**.
- Questa soluzione è imposta da un problema di congruenza dei dati; non essendo terminata **A1**, se il monitor **A** viene liberato, un nuovo processo può trovare **la struttura dei dati del monitor non consistente**.

CHIAMATE INNESTATE A MONITOR

Monitor A Monitor B

```
...
public void A1()
{
    ...
    B.B1;
    ...
}

public void A2()
{
    ...
    B.B2;
}
```

```
...
public void B1()
{
    ...
    X.wait;
    ...
}

public void B2()
{
    ...
    X.signal;
    ...
}
```

The diagram illustrates the state of two monitors, A and B, during a nested call. Monitor A contains code for procedures A1 and A2. A1 calls B1, which suspends the process on condition X. Monitor B contains code for procedures B1 and B2. B1 suspends the process on X. When B2 returns to A2, it signals X, allowing the process to resume at the end of A2. A red arrow points from A1 to B1, and a blue arrow points from B2 back to A2.

Soluzioni

1. proibire l'innestamento
2. in caso di sospensione, vengono liberati tutti i monitor interessati dalla catena di chiamate. -> difficile implementazione:
 - necessita di garantire la consistenza delle variabili del monitor prima di chiamate innestate;
 - al risveglio, il processo deve riacquisire tutti i monitor.
3. permettere procedure del monitor non mutuamente esclusive