I Monitor

- Rappresentano una primitiva di alto livello di supporto alla stesura di programmi concorrenti corretti
  - Brinch Hansen (1973), Hoare (1974)
- Un Monitor è una raccolta di dati, strutture dati e procedure raggruppate in un modulo o pacchetto
  - I processi possono invocare le procedure di un Monitor in ogni momento
  - I processi non possono accedere ai dati di un Monitor se non attraverso le procedure del Monitor
  - In un Monitor può essere attivo un solo processo alla volta
    - Un Monitor implementa la mutua esclusione grazie all'acquisizione (implicita) di un lock (mutex)

```
monitor monitor-name
{
  // shared variable
  declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     Initialization code (…)
  { … }
  }
}
```

// Simile ad una classe C++ o Java

Esempio: ipotetico Monitor C++

```
monitor class account {
private:
     int balance = 0;
public:
     void deposit(int amount) {
          balance = balance + amount;
     }
     void withdraw(int amount) {
          balance = balance - amount;
     }
};
```

I Monitor

- Il lock per la mutua esclusione non è sufficiente
- Esempio
  - per il produttore/consumatore, in precedenza abbiamo usato i semafori per
    - mettere a riposo i thread in attesa che una condizione cambi (per esempio, un produttore che aspetta che un buffer venga svuotato),
    - per risvegliare un thread quando una particolare condizione è cambiata (per esempio, un consumatore che segnala di aver effettivamente svuotato un buffer)
- I Monitor supportano tali funzionalità con un costrutto esplicito noto come variabile di condizione

Sincronizzazione con Monitor

- Per l'elaborazione concorrente un monitor deve fornire strumenti di sincronizzazione
  - Supponiamo che un processo invochi un monitor e, all'interno del monitor, deve bloccarsi fino a che si verifichi una certa condizione
    - Esempio, produttori che trovano buffer pieno
  - E' necessario un supporto affinché il processo quando si blocca rilascia il monitor in modo da consentire ad altri processi l'ingresso
  - Quando la condizione sarà soddisfatta ed il monitor di nuovo disponibile, al processo deve essere consentito di sbloccarsi e di rientrare nel monitor nel punto in cui è era stato sospeso

Monitor: funzionalità generali

- Un Monitor definisce un lock (implicito) e zero o più variabili di condizione per gestire l'accesso concorrente ai dati
  - Usa il lock per assicurare che solo un processo è attivo nel monitor in ogni momento
  - Il lock fornisce anche la mutua esclusione per i dati condivisi
  - Le variabili di condizione permettono ai processi di bloccarsi nella sezione critica, rilasciando il loro lock e, allo stesso tempo, bloccando il processo
- Operazioni del Monitor
  - Incaspula i dati condivisi da proteggere
  - Acquisisce il lock all'inizio
  - Opera sui dati condivisi
  - Rilascia il lock temporaneamente se non può completare
  - Riacquisisce il lock appena può continuare
  - Rilascia il lock alla fine

Sincronizzazione con Monitor: variabili di condizione

- La sincronizzazione è supportata nel monitor con l'uso delle variabili di condizione, contenute ed accessibili solo nel monitor
  - Sono uno tipo speciale tipo di dato sul quale sono possibili due operazioni
    - `cond_wait(c)`
      - Sospende sulla variabile di condizione c l'esecuzione del processo che la invoca
    - `cond_signal(c)`
      - Riprende l'esecuzione di qualche processo bloccato dopo una `cond_wait` sulla stessa variabile di condizione
- Osservazione
  - le operazioni `cond_wait()` e `cond_signal()` sono diverse rispetto alle `wait()` e `signal()` dei semafori
  - Se un processo invoca `cond_signal()` e non c'è nessuno in attesa sulla variabile di condizione il segnale è perso

I Monitor: variabili di condizione

- **`condition x;`**
  - **`x.cond_wait()`** un processo che la invoca si blocca fino ad una **`x.cond_signal()`**
    - Il lock mantenuto dal processo è rilasciato atomicamente quando il processo si blocca
  - **`x.cond_signal()`** risveglia un processo (se esiste) che ha invocato **`x.cond_wait()`**
    - Se non ci sono state **`x.cond_wait()`** sulla variabile, non ha effetto

Esempio: uso delle variabili di condizione

- Consideriamo P_1 e P_2 che eseguono due istruzioni S_1 e S_2 con il vincolo che S_1 sia eseguita prima di S_2
  - Creiamo un monitor con due procedure F_1 and F_2 invocate da P_1 e P_2, rispettivamente
  - Una variabile di condizione "x"
  - Una variabile booleana "done" inizializzata a false
  - **F1:**

```
S1;
done = true;
x.signal();
```

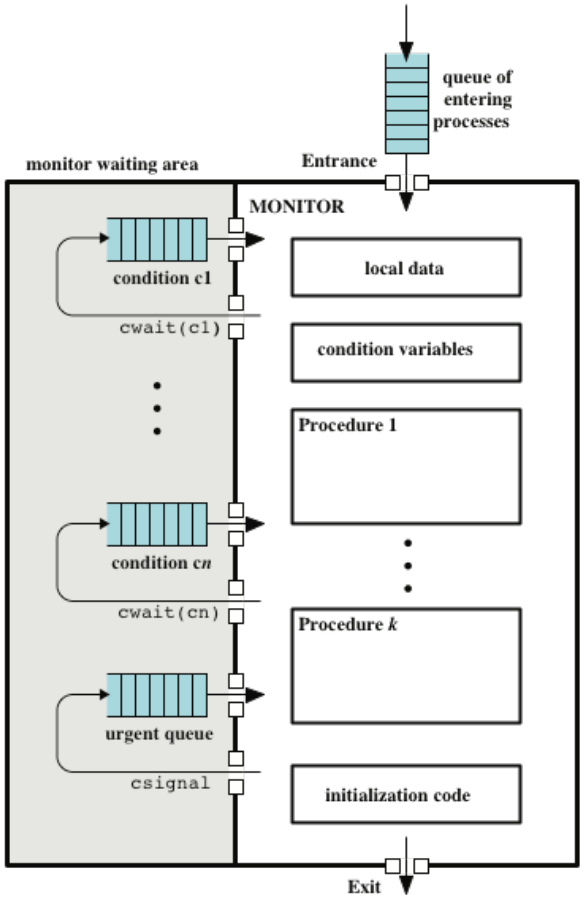
  - **F2:**

```
if done = false
   x.wait()
S2;
```

Monitor: regole per le variabili di condizione

- Dopo una **`cond_signal`** è necessario avere solo un processo attivo nel monitor
  - Abbiamo bisogno di una regola per decidere chi risvegliare
- Hoare
  - Eseguiamo il processo appena risvegliato, sospendendo l'altro (che ha invocato **`cond_signal`**)
- Brinch Hansen
  - Il processo che invoca **`cond_signal`** deve uscire immediatamente
    - **`cond_signal`** può apparire solo come istruzione finale di una procedura del monitor
  - Concettualmente più semplice e più facile da implementare
  - Dopo la **`cond_signal`**, lo scheduler seleziona uno solo dei processi in attesa
- Mesa
  - Il processo, P, che invoca **`cond_signal`** continua
  - Il processo in attesa, bloccato, è spostato tra i processi ready e dunque non è detto sia subito schedulato ed eseguito
  - Le implementazioni moderne delle variabili di condizione si basano su questa semantica

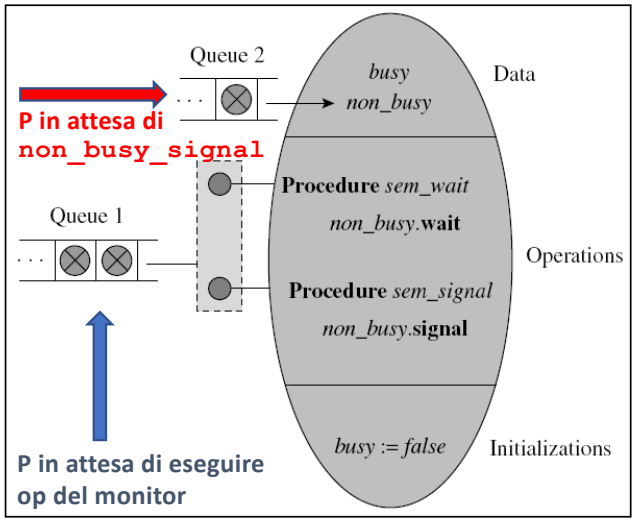
Struttura di un Monitor



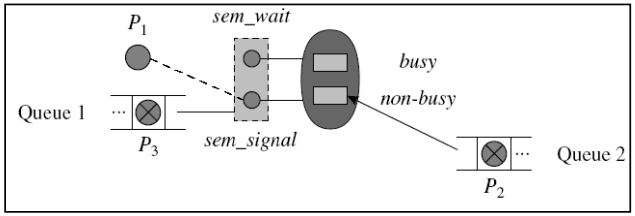
Semaforo Binario con Monitor

```
type Sem_Mon_type = monitor
  var
     busy:boolean;
     non_busy: condition;
  procedure sem_wait;
  begin
          if busy = true then non_busy.cond_wait;
          busy := true;
  end
  procedure sem_signal;
  begin
          busy := false;
          non_busy.cond_signal;
  end
  begin { initialization }
          busy := false;
  end
```

```
var binary_sem : Sem_Mon_type;
begin
Parbegin
repeat                    repeat                    repeat
  binary_sem.sem_wait;      binary_sem.sem_wait;      binary_sem.sem_wait;
  { Critical Section }      { Critical Section }      { Critica lSection }
  binary_sem.sem_signal;    binary_sem.sem_signal;    binary_sem.sem_signal;
  { Remainder of            { Remainder of            { Remainder of
    the cycle }               the cycle }               the cycle }
forever;                  forever;                  forever;
Parend;
end.
Process P1                Process P2                Process P3
```



Monitor per il semaforo binario



P_1 accede in SC, P_2 cerca di eseguire sem_wait, P_3 prova ad eseguire sem_wait prima che P_1 finisca di eseguire sem_signal

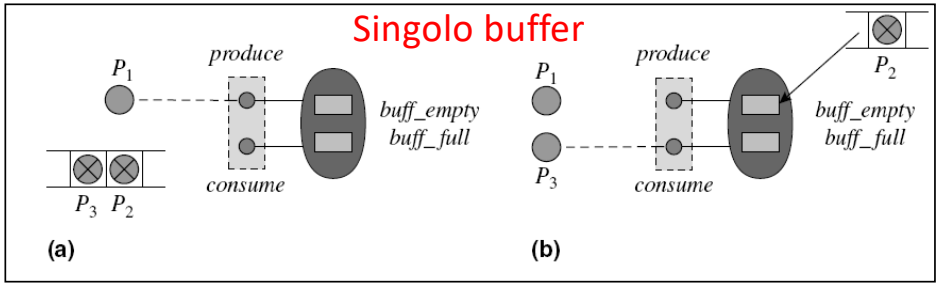
Esempio: Produttori-Consumatori con Monitor

```
type Bounded_buffer_type = monitor
    const
        n = . . .;                          { Number of buffers }
    type
        item = . . .;
    var
        buffer : array [0..n–1] of item;
        full, prod_ptr, cons_ptr : integer;
        buff_full : condition;
        buff_empty : condition;
    procedure produce (produced_info : item);
    begin
        if full = n then buff_empty.wait;
        buffer [prod_ptr] := produced_info;          { i.e., Produce }
        prod_ptr := prod_ptr + 1 mod n;
        full := full + 1;
        buff_full.signal;
    end;
    procedure consume (for_consumption : item);
    begin
        if full = 0 then buff_full.wait;
        for_consumption := buffer[cons_ptr];         { i.e., Consume }
        cons_ptr := cons_ptr + 1 mod n;
        full := full–1;
        buff_empty.signal;
    end;
    begin { initialization }
        full := 0;
        prod_ptr := 0;
        cons_ptr := 0;
end;
```

Attendere quando no buffer vuoti

Per attendere quando no buffer pieni

Esempio: Produttori-Consumatori con Monitor




```
begin
var B_buf : Bounded_buffer_type;

Parbegin
  var info : item;                var info : item;                var area : item;
  repeat                          repeat                          repeat
      info := . . .                   info := . . .                   B_buf.consume (area);
      B_buf.produce (info);           B_buf.produce (info);           { Consume area }
      { Remainder of                  { Remainder of                  { Remainder of
        the cycle }                     the cycle }                     the cycle }
  forever;                        forever;                        forever;
Parend;
end.
```

Producer P_1 *Producer P_2* *Consumer P_3*

```
type Bounded_buffer_type = monitor
    const
        n = . . .;                              { Number of buffers }
    type
        item = . . .;
    var
        buffer : array [0..n–1] of item;
        full, prod_ptr, cons_ptr : integer;
        buff_full : condition;
        buff_empty : condition;
    procedure produce (produced_info : item);
    begin
        if full = n then buff_empty.wait;
        buffer [prod_ptr] := produced_info;     { i.e., Produce }
        prod_ptr := prod_ptr + 1 mod n;
        full := full + 1;
        buff_full.signal;
    end;
    procedure consume (for_consumption : item);
    begin
        if full = 0 then buff_full.wait;
        for_consumption := buffer[cons_ptr];    { i.e., Consume }
        cons_ptr := cons_ptr + 1 mod n;
        full := full–1;
        buff_empty.signal;
    end;
    begin { initialization }
        full := 0;
        prod_ptr := 0;
        cons_ptr := 0;
end;
```

Ancora sulla semantica Mesa

- Le implementazioni moderne delle variabili di condizione sono baste sulla semantica Mesa
- Ciò pone un problema
  - Consideriamo di nuovo il codice

L'uso dell'If può generare dei bug

Soluzione ?

Sostituire If con while !!!

```
procedure produce (produced_info : item);
begin
    if full = n then buff_empty.wait;
    buffer [prod_ptr] := produced_info;          { i.e., Produce }
    prod_ptr := prod_ptr + 1 mod n;
    full := full + 1;
    buff_full.signal;
end;
procedure consume (for_consumption : item);
begin
    if full = 0 then buff_full.wait;
    for_consumption := buffer[cons_ptr];         { i.e., Consume }
    cons_ptr := cons_ptr + 1 mod n;
    full := full–1;
    buff_empty.signal;
end;
begin { initialization }
    full := 0;
    prod_ptr := 0;
    cons_ptr := 0;
end;
```

Problema del barbiere addormentato

- In un negozio di barbiere c'è
  - Un barbiere
  - N sedie per i clienti in attesa
  - 1 poltrona da barbiere
- Specifiche
  - Se non ci sono clienti, il barbiere si addormenta sulla poltrona
  - Quando arriva, un cliente sveglia il barbiere e si fa tagliare i capelli sulla poltrona
  - Se arriva un cliente mentre il barbiere sta tagliando i capelli ad un altro cliente, il cliente attende su una delle sedie libere
  - Se tutte le sedie sono occupate, il cliente, contrariato, se ne va!