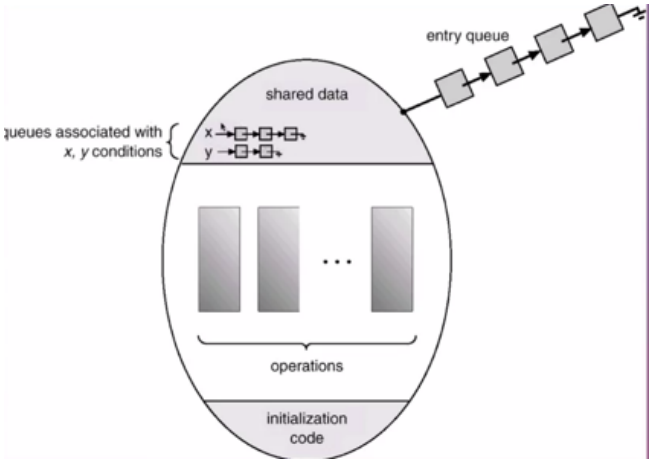
💙

Lezione 7.4



Monitor

Lo scorretto uso dei semafori non può essere sempre identificato in quanto si manifestano solo in presenza di determinate sequenze

E neanche l'uso dei semafori garantisce che non si verifichino errori di sincronizzazione

Qualche esempio:

1. Supponiamo che venga capvolto l'ordine di signal() e di mutex() in questo modo:

```
signal(mutex);

    // Sezione critica

wait(mutex);
```

   In questa soluzione, molti processi possono eseguire la loro sezione critica in contemporanea, violando la mutua esclusione

2. Ipotizziamo che un processo sostituisca signal(mutex) con wait(mutex)

```
wait(mutex);

   // Sezione critica

wait(mutex);
```

   In questo caso si genera uno stallo, un *deadlock*

3. Ipotizziamo che un processo ometta wait(mutex),signal(mutex) o entrambi: si andrebbe contro il principio della mutua esclusione oppure si creerebbe un deadlock

Per rimediare a questi errori sono stati creati i monitor

Monitor: E’ una primitiva di sincronizzazione ad alto livello. E’ un **tipo di dato astratto** e un **modulo software** (ADT: incapsula i dati mettendo a disposizione una serie di operatori per operare su di essi; queste funzioni sono indipendenti dalla specifica del tipo di dato) che comprende un insieme di operazioni definite dal programmatore che, all’interno del monitor, sono contraddistinte dalla mutua esclusione.

La rappresentazione di un tipo monitor non può essere utilizzata direttamente dai vari processi. Per questo, una funzione che è stata definita all’interno del tipo monitor può utilizzare solo le variabili dichiarate localmente nel monitor e quelle relative ai variabili formali.

Allo stesso modo, le variabili locali di un monitor possono accedere solo alle procedure locali

Quindi contiene: una o più procedure, una sequenza di inizializzazione e dati locali

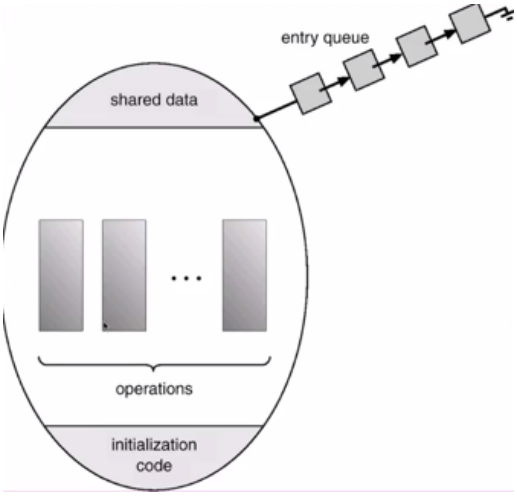
Sintassi:

```
monitor monitor_nome {
    /* dichiarazione di variabili condivise*/

    function P1(...) {
        ...
    }

    function P2(...) {
        ...
    }
    .
    .
    .
    function Pn(...) {
        ...
    }

    initialization_code(...) {
        ...
    }
}
```


Le variabili locali sono accessibili solo dalle procedure del monitor e non dalle procedure esterne

Un processo entra nel monitor chiamando una delle sue procedure

Questo modulo assicura che ci sarà solo un processo attivo all’interno del monitor: ogni altro processo he ha chiamato il monito sarà sospeso nell’attesa che questo diventi disponibile. Si garantisce, così, la mutua esclusione senza doverla codificare esplicitamente



Variabili condition

Un monitor deve contenere degli strumenti di sincronizzazione

Ad esempio: si supponga che un processo chiami il monitor e che mentre è al suo interno sia sospeso finché non si verifica una certa condizione. E' necessario fare in modo che il processo sospeso rilasci il monitor in modo che altri processi possano entrare

Quando, in seguito, la condizione viene soffisfatta e il monitor è nuovamente disponibile, il processo deve essere riattivato e deve poter rientrare nel monitor nello stesso punto in cui era stato sospesi

Un monito fornisce la sincronizzazione mediante l'uso di **variabili di condizione** accessibili solo dall'interno del monitor

Quindi, un programmatore che deve scriver un proprio schema di sincronizzazione può definire una o più variabili condizionali tramite il costrutto **condition**

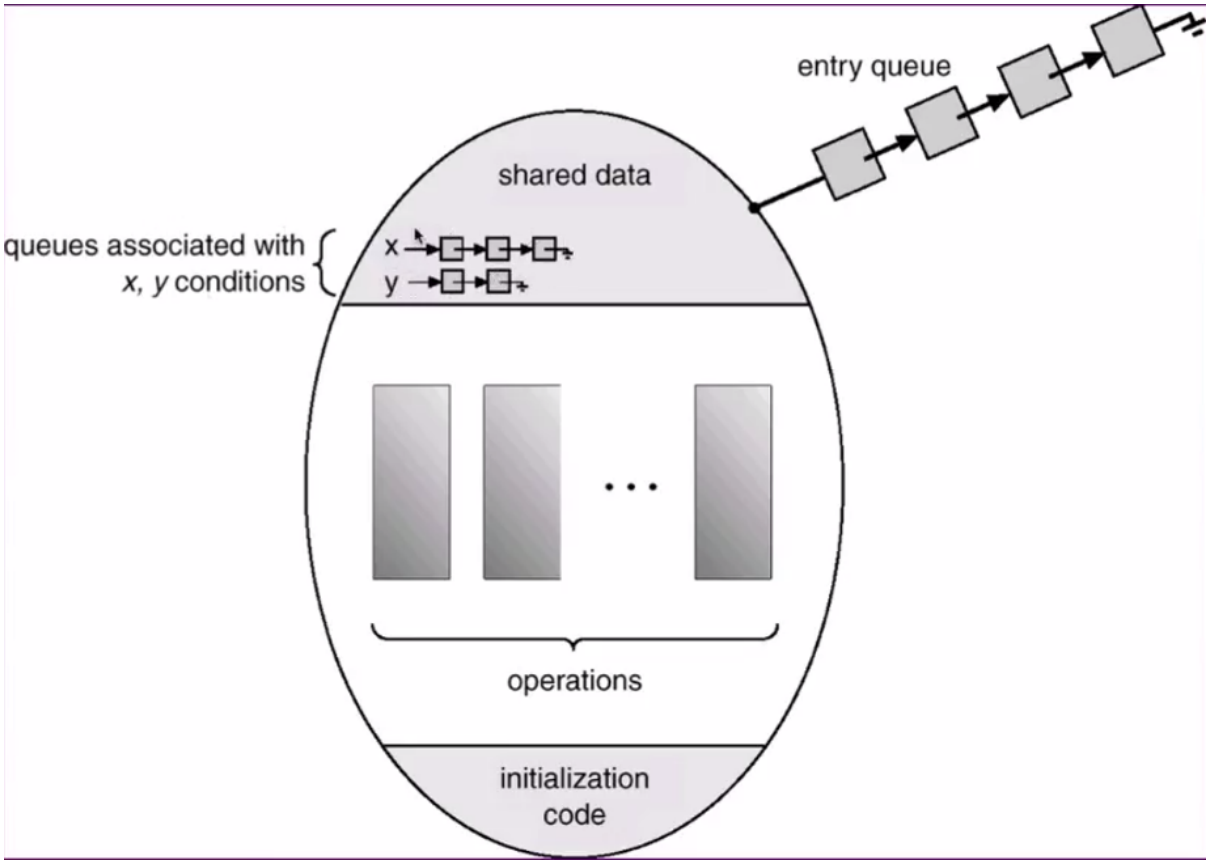
Ex. `condition x, y;`

Le uniche operazioni che possono essere eseguite su una variabile di tipo condition sono **wait() e signal()**

L'operazione → x.wait(): implica che il processo che la invoca rimanga sospeso finchè un altro proceso non invochi
l'operazione → x.signal(): risveglia esattamente un processo sospeso

Le operazioni di wait() e signal() che vengono svolte sulle variabili condition sono differenti da quelle dei semafori: se non esistono processi sospesi l'operazione signal() non ha alcun effetto, vale a dire che lo stato di x resta immutato come se l'operazione non fosse stata eseguita. Al contrario, la signal() effettuata su un semaforo cambia sempre lo stato del semaforo

Schema di un monitor con variabili condition



Si supponga che quando un processo P invoca l'operazione x.signal(), esista un processo sospeso Q associato alla variabile z di tipo condition. chiaramente, se al processo sospeso Q si permette di riprendere l'esecuzione, P sarà costretto ad attendere. Esistono quindi due possibilità:

- **segnalare ed attendere**: P attende che Q lasci il monitor o attenda su un'altra variabile condition
- **segnalare e proseguire**: Q attende che P lasci il monitor o attenda su un'altra variabile condition

Entrambe sono soluzioni ragionevoli, ma c'è da tener presente che nel caso P continuasse, la condizione attesa da Q potrebbe non valere più al momento in cui quest'ultimo riprende l'esecuzione

Soluzione al problema dei cinque filosofi per mezzo di monitor

Vincolo: un filosofo può presente le sue bacchette solo quando sono entrambe disponibili

Per questo si introduce la struttura dati:

```
enum {THINKING, HUNGRY, EATING} state [5];
```

Il filosofo può impostare `state[i] = EATING;` solo se i suoi due vicini non stanno mangiando:

```
(state[(i + 4) % 5] != EATING) && (state[(i + 1) % 5] != EATING)
```

Inoltre, occorre dichiarare la struttura dati§:

```
condition self[5];
```

che permette al filosofo i di ritardare se stesso quando ha fame, ma non riesce ad avere le bacchette

```
monitor DiningPhilosophers {

    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i);
    void posa(int i);
    void verifica(int i);

    void inizializzazione() {
        for (int i = 0; i < 5; i++) stato[i] = pensa;
    }
    void pickup(int i) {
        state[i] = HUNGRY // cambia stato da pensa passa ad hungry
        test(i); // verifica se può prendere le bacchette a destra e a sinistra
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING; // reimposta il suo stato a thinking perché ha mangiato
        test((i + 4) % 5);
        test((i + 1) % 5); // controlla se quelli a destra e a sinistra hanno fame, per sbloccarli
    }

    void test(int i) { // verifica se alla sua sinistra o alla sua destta stanno mangiando
        if((state[(i + 4) % 5] != EATING) && //  se a destra e a sinistra non stanno mangiando, e lui vuole mangiare
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
            state[i] = EATING; // passa allo stato mangia
            self[i].signal(); // se era bloccato si libera
        }
    }
}
```


Possono esseri due filosofi attivi contemporaneamente

Un solo filosofo per volta sbloccherà gli altri

Spiegazione:

Ogni filosofo, prima di iniziare, deve richiamare la funzione pickup(): ciò potrebbe determinare il blocco del processo.
Se viene completata con successo l’operazione, il filosofo può mangiare; terminata la sua attività, il filosofo invoca putdown() e ricomincia a pensare

Il filosofo esegue le due operazioni pickup() e putdown() come segue:

```
DiningPhilosophers.pickup(i);
  ...
  eat
  ...
DiningPhilosophers.putdown(i);
```

E’ semplice dimostrare che questa soluzione assicura che due filosofi vicini non mangino contemporaneamente e che non si verifichino situazioni di stallo. E’ necessario nota che che, però, un filosofo può attendere indefinitivamente.