

Costrutti linguistici per la sincronizzazione

I semafori costituiscono un meccanismo **molto potente** per la sincronizzazione dei processi. Tuttavia, il suo uso può risultare troppo “a basso livello” (linguaggio assembler).

Possibilità di errori

Esempio: mutua esclusione

- *scambio tra wait e signal:*

```
    signal(mutex);  
        <A>;  
    wait(mutex);
```

più processi possono operare nella sezione critica.

- *utilizzo erraneo di wait e signal:*

```
    wait(mutex);  
        <A>;  
    wait(mutex); -> deadlock.
```

- Omissione di **wait**(mutex) o di **signal**(mutex)
- Per ovviare a problemi di questa natura si sono introdotti costrutti linguistici “a più alto livello” .

Regioni critiche semplici e condizionali

Regione critica semplice:

```
shared T  v; /* variabile condivisa */  
region v do {S1, S2, ....Sn };
```

- La sequenza di statement S1, S2,Sn ha accesso alla variabile **shared** v.
- Il compilatore può verificare che la variabile **v** sia usata solo all'interno della regione critica e può implementare correttamente la mutua esclusione.

Proprietà

1. Il compilatore inserisce automaticamente **wait** e **signal**

shared T v;	$\Rightarrow$	semaphore mutex
		mutex.value=1;
region v do {S };	$\Rightarrow$	wait (&mutex);
		S;
		signal(&mutex);

2. Il compilatore controlla che i processi accedano a **v solo entro la regione critica**

3. Il compilatore può riconoscere possibili *deadlock*:

```
shared T v1, v2;  
/* processo P1: */  
{ region v1 do  
  { region v2 do  
    { ...  
  }  
}  
}
```

```
shared T v1, v2;  
/* processo P2: */  
{ region v2 do  
  { region v1 do  
    { ...  
  }  
}  
}
```

```
shared struct
{
    int top;
    messaggio stack[N];
}pila;
pila.top=0; /*valore iniziale*/
```

```
void inserimento(messaggio x)
{
    region pila do
    { if (pila.top==N-1)
        /*pila piena*/
        else
        { top++;
          pila.stack[top]=x;
        }
    }
}
```

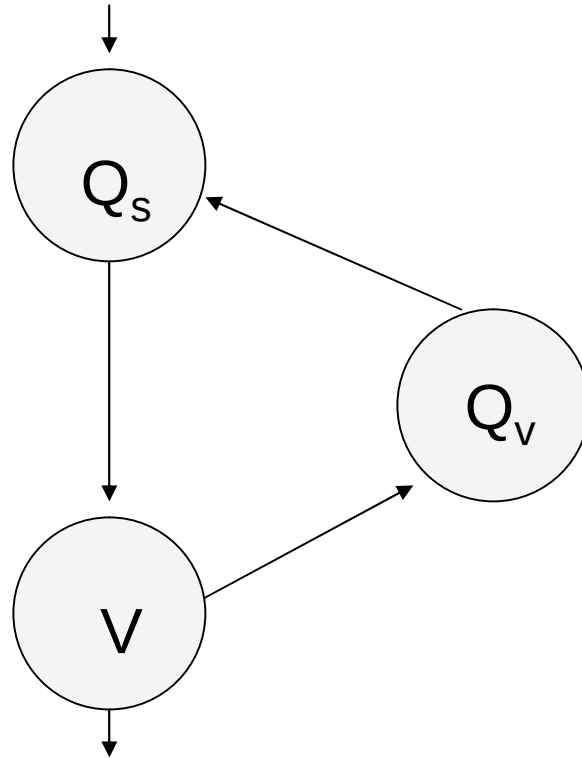
```
messaggio prelievo()
{
    messaggio x;
    region pila do
    { if (pila.top==0)
        /*pila vuota*/
        else
        { top--;
          x=pila.stack[top];
        }
    }
    return x;
}
```

Regione critica condizionale

region v when B do {S1; S2;Sn;}

E' usata per **ritardare** il completamento di una regione critica fino a che non si verifica una condizione.

- Quando il processo entra nella regione critica viene valutata la condizione booleana **B**:
 - **se B è vera**, la regione critica è completata eseguendo S1; S2;Sn ;
 - **se B è falsa**, il processo libera la regione critica e si sospende in una coda associata alla variabile **v**.
- Gli operandi dell'espressione booleana **B** sono componenti della struttura dati **v**.



Q_v = coda associata
alla variabile v

Q_s = coda
associata al
semaforo

Quando la regione critica è libera, un altro processo può accedervi e completare (eventualmente) la regione critica

Questo processo può avere modificato la struttura di dati v e reso vera la condizione **B**

Pertanto i processi sospesi in Q_v devono **rientrare** nella sezione critica e **rivalutare B**

- Il programmatore non controlla l'ordine con il quale i processi hanno accesso alla risorsa comune.
- Tutti i processi sospesi in **Qv** vengono trasferiti in **Qs** e il primo per il quale è vera la condizione di sincronizzazione, ha accesso alla risorsa.
- Non è possibile dare il controllo ad un particolare processo.
- Problemi di “starvation” risolti dando priorità ai processi della coda **Qv** rispetto a **Qs**.
- Strumento adatto per sistemi con poche interazioni (“loosely connected problems”)

```
shared struct
{      int testa, coda, cont;
      messaggio buffer[N];
}mailbox;
mailbox.testa=0; /*valore iniziale*/
mailbox.coda=0;
mailbox.cont=0;
```

```
void send(messaggio x)
{  region mailbox when cont<N do
  {  mailbox.buffer[coda]=x;
    coda=(coda+1)%N;
    cont++;
  }
}
```

```
messaggio receive()
{  messaggio x;
  region mailbox when cont>0 do
  {  x=mailbox.buffer[testa];
    testa=(testa+1)%N;
    cont--;
  }
  return x;
}
```

Realizzazione della sezione critica condizionale tramite semafori:

region v when B do {S1; S2;Sn;}

```
semaphore mutex;  
semaphore synch  
int cont=0;  
mutex.value=1;  
synch.value=0;  
wait(&mutex);  
while(!B)  
{  cont ++;  
    signal(&mutex);  
    wait (&synch);  
    wait (&mutex);  
}  
S;  
while (cont!=0)  
{  signal (&synch);  
    cont--;  
}  
signal (mutex);
```

Vantaggi ottenibili con le regioni critiche semplici e condizionali

a) maggiore chiarezza nel programma

b) controllo a tempo di compilazione:

- il compilatore può controllare che l'accesso a variabili *shared* **avvenga solo entro le regioni critiche.**
- il compilatore provvede direttamente ad assicurare la **mutua esclusione**, evitando così l'uso diretto dei semafori da parte del programmatore

Problemi

- I processi accedono direttamente alle variabili comuni (entro le regioni critiche)
- accesso non controllato alle risorse del sistema
- i processi realizzano direttamente l'algoritmo di scheduling