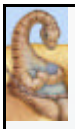
Capitolo 6: Sincronizzazione dei processi



Capitolo 6: Sincronizzazione dei processi

- Introduzione
- Problema della sezione critica
- Soluzione di Peterson
- Hardware per la sincronizzazione
- Semafori
- Problemi tipici di sincronizzazione
- Monitor
- Esempi di sincronizzazione
- Transazioni atomiche

Processo produttore

```
while (true) {

        /*  produce un elemento in appena_Prodotto  */
         while (contatore == DIM_BUFFER)
                    ; // non fa niente
                buffer [inserisci] = appena_Prodotto;
                inserisci = (inserisci + 1) % DIM_BUFFER;
                contatore++;
}
```

Processo consumatore

```
while (true)  {
        while (contatore == 0)
                ; // non fa niente
                da_Consumare =  buffer[preleva];
                preleva = (preleva + 1) % DIM_BUFFER;
                 contatore--;

                    /*  Consuma un elemento in da_Consumare */
 }
```

Algoritmo per il processo P_i

```
while (true) {
        flag[i] = true;
        turno = j;
        while ( flag[j] && turno = = j);

            sezione critica

        flag[i] = false;

        sezione non critica

 }
```


Istruzione TestAndSet

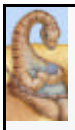
```
boolean TestAndSet (boolean *obiettivo)
 {
     boolean valore = *obiettivo;
     *obiettivo = true;
     return valore:
 }
```

Realizzazione di mutua esclusione tramite TestAndSet()

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* non fa niente

                //    sezione critica

        lock = false;

                //      sezione non critica

}
```

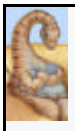
Definizione di Swap()

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp;
}
```



Realizzazione di mutua esclusione con Swap()

```
while (true)  {
        chiave = true;
        while ( chiave = = true)
                Swap (&lock, &chiave );

                //    sezione critica

        lock = false;

                //      sezione non critica

}
```



Semafori come generale strumento di sincronizzazione

- Semafori contatore – valore numerico illimitato
- Semafori binari – valore è 0 o 1
  - Detti anche lock mutex
- Garantiscono la mutua esclusione
  - Semaforo S; // inizializzato a 1
  - wait (S);

    sezione critica

    signal (S);

Uso dei semafori senza attesa attiva

- Uso di wait():

```
wait (S){
        valore--;
        if (valore < 0) {
                aggiungi questo processo a S->lista;
                block();  }
 }
```

- Uso di signal():

```
signal (S){
        valore++;
        if (valore <= 0) {
                togli un processo P da S->lista;
                wakeup(P);  }
}
```

Produttori e consumatori con memoria limitata (Cont.)

- Struttura generale del processo produttore

```
while (true)  {

        //   produce un elemento

    wait (vuote);
    wait (mutex);

        //  inserisce un elemento nel buffer

     signal (mutex);
     signal (piene);
}
```

Produttori e consumatori con memoria limitata (Cont.)

- Struttura generale del processo consumatore

```
while (true) {
        wait (piene);
        wait (mutex);

              // rimuovi un elemento dal buffer

        signal (mutex);
        signal (vuote);

              // consuma l'elemento

}
```


Problema dei lettori-scrittori (Cont.)

- Struttura generale di un processo scrittore:

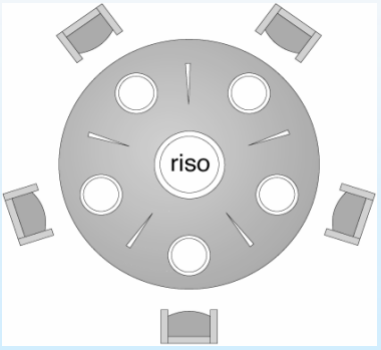
```
while (true) {
        wait (scrittura) ;

            //    esegui l'operazione di scrittura

        signal (scrittura) ;
}
```

Problema dei lettori-scrittori (Cont.)

- Struttura generale di un processo lettore

```
while (true) {
        wait (mutex) ;
        numlettori ++ ;
        if (numlettori = = 1)  wait (scrittura) ;
        signal (mutex)

                // esegui l'operazione di lettura

        wait (mutex) ;
      numlettori - - ;
        if (numlettori == 0)  signal (scrittura) ;
        signal (mutex) ;
}
```


Problema dei cinque filosofi



Problema dei cinque filosofi

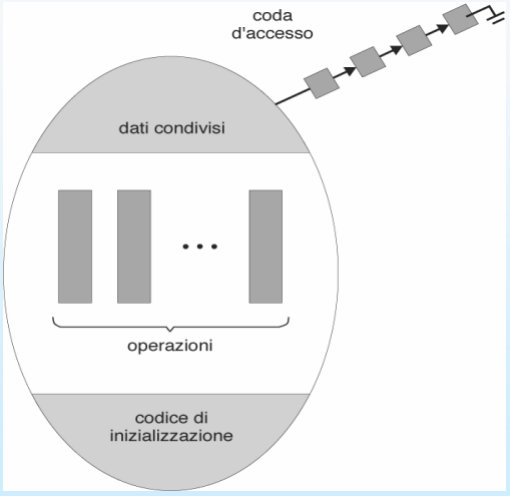
- Struttura del filosofo *i*:

```
While (true)  {
        wait ( bacchetta[i] );
         wait (bacchetta[ (i + 1) % 5] );

              //  mangia

         signal (bacchetta[i] );
         signal (bacchetta[ (i + 1) % 5] );

             //  pensa

}
```

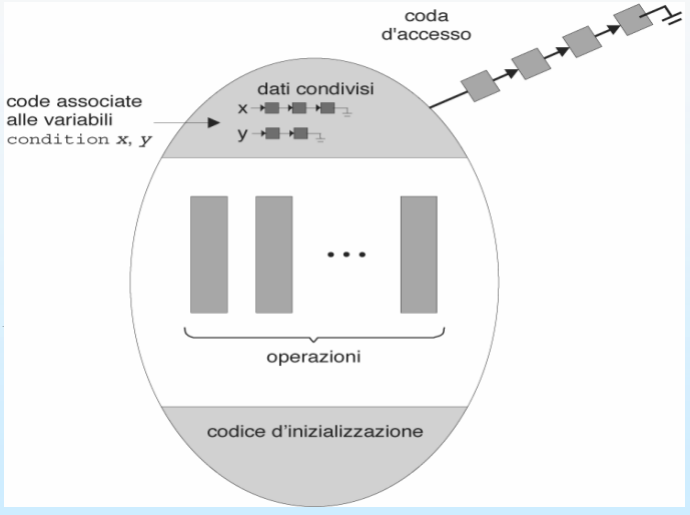

Sintassi di un monitor

```
monitor nome_monitor
{
   // dichiarazioni di variabili condivise
   procedure P1 (…) { …. }
            …

   procedure Pn (…) {……}

    Codice d'inizializzazione ( ….) { … }
            …
   }
}
```

Schema di un monitor



Monitor con variabili condition



Soluzione al problema dei cinque filosofi

```
monitor FC
  {
    enum {pensa, affamato, mangia) stato [5] ;
    condition auto [5];

    prende (int i) {
          stato[i] = affamato;
         verifica(i);
          if (stato[i] != mangia)
            auto [i].wait;
    }

     posa  (int i) {
          stato[i] = pensa;
               // verifica i commensali di destra e sinistra
          verifica((i + 4) % 5);
          verifica((i + 1) % 5);
      }
```

Soluzione al problema dei cinque filosofi (Cont.)

```
    verifica  (int i) {
          if ( (stato[(i + 4) % 5] != mangia) &&
          (stato[i] == affamato) &&
          (stato[(i + 1) % 5] != mangia) ) {
              stato[i] = mangia ;
               auto[i].signal () ;
           }
     }

     codice di inizializzazione() {
          for (int i = 0; i < 5; i++)
          stato[i] = pensa;
     }
}
```

Soluzione al problema dei cinque filosofi (Cont.)

- Ciascuno dei filosofi *i* invoca le operazioni pickup() e putdown() nella sequenza:

```
fc.prende (i)

    mangia

fc.posa (i)
```


Realizzazione di un monitor

- Per ciascuna variabile condizionale ***x***, abbiamo:

```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

- L'operazione x.wait è realizzabile come:

```
x-contatore++;
if (prossimo-contatore > 0)
    signal(prossimo);
else
    signal(mutex);
wait(x-sem);
x-contatore--;
```

Realizzazione di monitor

- L'operazione x.signal è realizzabile come:

```
if (x-contatore > 0) {
    prossimo-contatore++;
    signal(x-sem );
    wait(prossimo);
    prossimo-contatore--;
}
```


Sequenza d'esecuzione 1: T_0 poi T_1

T_0	T_1
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Sequenza d'esecuzione 2: sequenza d'esecuzione concorrente serializzabile

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Sequenza d'esecuzione 3 con il protocollo basato sulla marcatura temporale

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Fine del Capitolo 6