

Sincronizzazione dei processi

234

Background

- Accesso concorrente a dati condivisi può portare ad avere dati inconsistenti, dovuto a:
 - Scheduling con prelazione in monoprocesso
 - Accesso parallelo alla memoria in multiprocesso
- Mantenere consistenza dei dati necessita di meccanismi per assicurare l'esecuzione ordinata dei processi concorrenti
- Supponiamo di realizzare un **processo/thread produttore** e un **processo/thread consumatore** che condividono una zona di memoria per memorizzare degli elementi.
 - Utilizza una variabile condivisa intera **count** che tiene traccia di quanti elementi sono pronti per il consumatore. Inizialmente è impostato a 0.
 - Viene **incrementato dal produttore** quando un nuovo elemento è disponibile e viene **decrementato dal consumatore** quando consuma un elemento.

235

Thread Produttore

```
while (true)
{
    /* produce un elemento e lo mette in
       nextProduced */
    while (count == BUFFER_SIZE)
        ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

Thread Consumatore

```
while (true)
{
    while (count == 0)
        ; // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    /* consuma l'elemento in nextConsumed */
}
```

Race Condition

- `count++` potrebbe essere implementata come
`register1 = count`
`register1 = register1 + 1`
`count = register1`
- `count--` potrebbe essere implementata come
`register2 = count`
`register2 = register2 - 1`
`count = register2`
- Consideriamo questa possibile esecuzione con inizialmente "count = 5":
 - S0: *produttore* esegue `register1 = count` {register1 = 5}
 - S1: *produttore* esegue `register1 = register1 + 1` {register1 = 6}
 - S2: *consumatore* esegue `register2 = count` {register2 = 5}
 - S3: *consumatore* esegue `register2 = register2 - 1` {register2 = 4}
 - S4: *produttore* esegue `count = register1` {count = 6}
 - S5: *consumatore* esegue `count = register2` {count = 4}
- Questo porta ad uno stato inconsistente in quanto lo stato finale dopo `count++` e `count--` dovrebbe essere `count=5` indipendentemente dall'ordine
- Le istruzioni `count++` e `count--` sono dette **sezioni critiche** dei due processi/thread

Race Condition

- In generale si ha una race condition quando vengono eseguite operazioni di lettura e modifica di un dato condiviso da parte di due o più thread.
- Esempio conto bancario (1000 €)
- **Prelievo**
 - leggo stato conto
 - decrementa di 500 €
 - salvo stato conto
- **Deposito**
 - leggo stato conto
 - incremento di 5000 €
 - salvo stato conto
- Deposito e Prelievo eseguiti concorrentemente, lo stato finale del conto può essere: **5500 €**, **500 €** o **6000 €**

Soluzione al problema della sezione critica

Proprietà delle sezioni critiche:

1. **Mutua esclusione** - Se il processo P_i è in esecuzione nella sua sezione critica allora nessun altro processo può essere in esecuzione nella propria sezione critica
2. **Progresso** - Se nessun processo è in esecuzione nella sezione critica ed esistono processi che vogliono entrare nella propria sezione critica allora la selezione di quale processo entrerà nella sezione critica non può essere postposta indefinitivamente
3. **Attesa limitata** - Deve esistere un limite nel numero di volte in cui altri processi entrano nella loro sezione critica dopo che un processo ha richiesto di entrare nella sezione critica e prima che la richiesta sia accordata
 - Assume che ogni processo venga eseguito a velocità non nulla.
 - Nessuna assunzione sulla velocità relativa degli N processi

Soluzione di Peterson

- Soluzione per 2 processi
- Assume che le istruzioni LOAD e STORE siano atomiche; quindi non possono essere interrotte.
- I due processi condividono le due variabili:
 - `int turn;`
 - `bool flag[2]`
- La variabile `turn` indica di chi è il turno per entrare nella sezione critica.
- L'array `flag` è usato per indicare se un processo è pronto ad entrare nella sezione critica. `flag[i] = true` implica che il processo P_i è pronto!

Algoritmo per processo P_i

process P_0

```
do {
    flag[0] = TRUE;
    turn = 1;
    while ( flag[1] && turn == 1)
        ;
    CRITICAL SECTION
    flag[0] = FALSE;
    NON CRITICAL SECTION
} while(true);
```

process P_1

```
do {
    flag[1] = TRUE;
    turn = 0;
    while ( flag[0] && turn == 0)
        ;
    CRITICAL SECTION
    flag[1] = FALSE;
    NON CRITICAL SECTION
} while(true);
```

- **Mutua esclusione:** se entrambi fossero nella sezione critica si avrebbe che $(\text{flag}[1] == \text{FALSE} \parallel \text{turn} == 0) \&\& (\text{flag}[0] == \text{FALSE} \parallel \text{turn} == 1)$
Ma $\text{flag}[0] == \text{flag}[1] == \text{TRUE}$ quindi deve valere $\text{turn} == 0 \&\& \text{turn} == 1$ (assurdo)
- **Progresso:** se P_0 e P_1 vogliono entrare sicuramente uno dei due entra (dipende dal valore di turn)
- **Attesa limitata:** dopo che P_0 è entrato e P_1 è in attesa se P_0 richiede nuovamente di entrare viene prima sbloccato P_1 .

Algoritmo del Fornaio (o di Lamport)

- Soluzione per N processi
- Simile a prendere numero dal fornaio per essere serviti
 - Prende numero
 - Aspetta di avere numero più piccolo
- Usa due vettori di N elementi **condivisi** tra gli N processi
 - **choosing[i]** booleano dove true indica che il processo P_i è nella fase di scelta del suo numero
 - **number[i]** indica il numero preso dal processo P_i per entrare nella sezione critica e 0 indica che il processo non vuole entrare in sezione critica

Algoritmo del Fornaio

```

process P(i) {
  do {
    choosing[i] = true;
    number[i] = 1 + max(number[0], ..., number[N-1]) } Doorway
    choosing[i] = false;

    for(j=0; j<N; j++) {
      while(choosing[j]) {}
      while(number[j] != 0 && (number[j], j) < (number[i], i)) {} } Backery
    }
    CRITICAL SECTION
    number[i] = 0;
    NON CRITICAL SECTION
  } while(true);
}

```

$(a,b) < (c,d)$
equivale a
 $a < c \parallel (a==c \ \&\& \ b < d)$

Algoritmo del Fornaio

- Nella fase di scelta del numero due o più processi possono prendere lo stesso numero, in questo caso verrà selezionato quello con identificatore minore
- **Mutua esclusione:** al termine del ciclo $(number[i], i) < (number[j], j)$ con $j=0..n-1$ e $j \neq i$, se due processi x e y fossero entrambi nella sezione critica si avrebbe $(number[x], x) < (number[y], y)$ e $(number[y], y) < (number[x], x)$ assurdo
- **Progresso:** se nessuno è nella sezione critica e più processi vogliono entrare sicuramente uno verrà scelto (avrà $(number[i], i)$ minore)
- **Attesa limitata:** i valori di $number[]$ aumentano sempre, se un processo è in attesa prima o poi $(number[i], i)$ sarà il più piccolo tra quelli presenti

Hardware per la sincronizzazione

- Molti sistemi forniscono un supporto hardware per le sezioni critiche
- Monoprocessori - possono disabilitare le interruzioni
 - Il codice in esecuzione puo' eseguire senza essere interrotto
 - Generalmente inefficiente su sistemi multiprocessori
- Le macchine moderne forniscono istruzioni atomiche speciali
 - Atomiche = non interrompibili
 - Due tipi: o controllano il valore e poi lo impostano
 - O scambiano il contenuto di due parole di memoria

Istruzione TestAndSet

- Definizione:

```
bool TestAndSet (bool *target)
{
    bool rv = *target;
    *target = TRUE;
    return rv;
}
```

- Eseguita dal processore senza poter essere interrotta

Soluzione usando TestAndSet

- variabile booleana condivisa **lock**, inizializzata a *false*.
- Soluzione:

```
do {  
    while ( TestAndSet (&lock ))  
        ;    /* do nothing */  
  
    //critical section  
  
    lock = false;  
  
    //non critical section  
  
} while (true);
```

Istruzione scambio

- Definizione:

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

- Eseguita dal processore senza poter essere interrotta

Soluzione usando Swap

- **lock** e' una variabile booleana condivisa e inizializzata a *false*; Ogni processo ha una variabile locale *key*.

- Soluzione:

```
do {  
    key = true;  
    while(key == true)  
        Swap(&lock, &key );  
    //critical section  
    lock = false;  
    //non critical section  
} while (true);
```

Soluzione con attesa limitata

- In entrambi i casi precedenti le soluzioni hardware viste garantiscono la mutua esclusione e progresso ma non l'attesa limitata
- Una soluzione è la seguente per il processo P_i ($i = 0..n-1$):

```
do {  
    waiting[i] = true;  
    key = true;  
    while(waiting[i] && key)  
        key = TestAndSet(&lock);  
    //CRITICAL SECTION  
    j = (i+1) % n;  
    while(j!=i && !waiting[j])  
        j = (j+1)%n;  
    if(j==i)  
        lock = false;  
    else  
        waiting[j] = false;  
    //NON CRITICAL SECTION  
}while (true);
```

waiting[i] indica se il processo P_i è in attesa di entrare in sezione critica (inizializzato a false)

Cerca il prossimo processo che è in attesa sul lock, seguendo ordine $i+1..n-1, 0, 1, \dots, i$ e se lo trova lo sblocca. Questo garantisce attesa massima di $n-1$ rilasci

Considerazioni

- Le soluzioni viste fino ad ora nei momenti di attesa per l'accesso alla sezione critica usano CPU per controllare ripetutamente il valore di una o più variabili
- Sono dette soluzioni «**busy waiting**» o ad «attesa attiva»
- In questi momenti di attesa attiva la CPU è usata inutilmente, potrebbe essere usata per eseguire altri programmi.
- Il sistema operativo in genere fornisce primitive di sistema per la realizzazione di sezioni critiche, in genere si chiamano **mutex** (da **mutual exclusion**) ed in genere sono implementati senza ricorrere alla attesa attiva.
- I sistemi operativi forniscono spesso anche gli **spinlock** che invece usano l'attesa attiva, utili quando il costo del cambio di contesto (dovuto al rilascio della CPU) è molto più costoso dell'esecuzione della sezione critica (es. poche istruzioni)

Semafori

- Strumento di sincronizzazione
- Semaforo S - variabile intera
- Due operazioni standard modificano S: **wait()** e **signal()**
 - Originariamente chiamate **P()** e **V()** proposti da Dijkstra come iniziali delle parole olandesi "proberen" ("verificare") e "verhogen" ("incrementare").
- Meno complicate
- Un semaforo può essere usato solo tramite queste due operazioni **indivisibili**
 - ```
wait (S) {
 while (S <= 0)
 ; // no-op
 S--;
}
```
  - ```
signal (S) {  
    S++;  
}
```

Semafori come strumenti generali per la sincronizzazione

- Semafori contatore - il valore intero non viene vincolato
- Semafori binari - il valore intero può essere solo 0 o 1; può essere più semplice da implementare
 - Conosciuti anche come `mutex locks`
- Si può implementare un semaforo binario con un semaforo contatore
- Fornisce la mutua-esclusione

```
Semaforo mutex=1;    //inizializzato a 1
wait(mutex);
Critical Section
signal(mutex);
```

Uso semafori

- I semafori contatore si possono usare per regolare accesso ad un numero **n** di risorse che vengono utilizzate singolarmente
 - Si inizializza il semaforo **S** ad **n**
 - Va chiamata **wait(S)** prima di prelevare una risorsa, se le risorse sono terminate ($S=0$) entra in attesa
 - Va chiamata **signal(S)** dopo aver terminato l'uso di una risorsa, che incrementa **S** e se qualche thread era bloccato in attesa sulla wait ora può continuare

Uso semafori

- Si vuole che l'esecuzione di **S1** di Thread1 preceda l'esecuzione di **S2** in Thread2
- Si può usare un semaforo binario **synch** inizializzato a 0

| Thread1: | Thread2: |
|----------------|---------------------|
| ... | ... |
| S1; | |
| signal(synch); |→ wait(synch); |
| ... | S2; |
| | ... |

Implementazione dei semafori

- Deve garantire che due processi non possano eseguire `wait()` e `signal()` sullo stesso semaforo allo stesso tempo
- Comunque la `wait()` esegue una attesa attiva (*busy wait*) che occupa la CPU inutilmente durante l'attesa che la sezione critica sia libera
- Le applicazioni possono passare molto tempo nelle sezioni critiche e quindi questa non è una buona soluzione.

Implementazione semafori senza busy waiting

- Ogni semaforo ha associata un coda di attesa.
- Due operazioni:
 - **block** - inserisce il processo che invoca l'operazione in una coda di attesa e il processo non entrerà più nella coda dei processi ready e quindi non sarà più eseguito.
 - **wakeup** - rimuove uno dei processi nella coda di attesa e lo mette nella coda dei processi ready.
- Struttura dati:

```
typedef struct {  
    int value;  
    struct process* list;  
} semaphore;
```

Implementazione semafori senza busy waiting (Cont.)

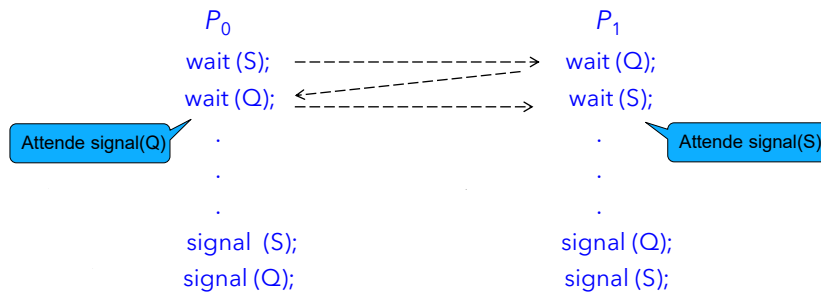
- Implementazione di *wait*:

```
wait(semaphore* S) {  
    S->value--;  
    if (S->value < 0) {  
        aggiunge il processo alla coda S->list;  
        block();  
    }  
}
```
- Implementazione di *signal*:

```
signal(semaphore* S) {  
    S->value++;  
    if (S->value <= 0) {  
        rimuove un processo P dalla coda S->list;  
        wakeup(P);  
    }  
}
```
- Devono essere eseguite in sezione critica (o disabilitando interruzioni in monoprocesso o usando spinlock in multiprocessore)

Stallo e Attesa indefinita

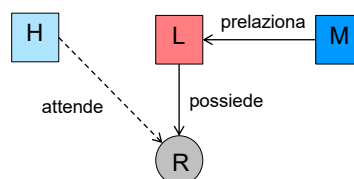
- Stallo (*deadlock*) – due o più processi aspettano indefinitamente un evento che può essere causato solo da uno dei processi in attesa
- Siano S e Q due semafori inizializzati a 1



- Attesa indefinita (*starvation*) – Un processo può non essere mai rimosso dalla coda di un semaforo dove è sospeso. Può accadere per esempio se coda di attesa gestita come LIFO.

Inversione di priorità

- Si ha quando un processo H ad alta priorità è in attesa di una risorsa R posseduta da un processo L a bassa priorità che viene prelazionato da un processo a più alta priorità $M < H$.
- M ritarda il rilascio della risorsa da parte di L e quindi ritarda l'esecuzione del processo ad alta priorità la cui eseguibilità dipende da un processo a priorità più bassa M .



Inversione di priorità

- Questo non accade se sono presenti solo due livelli di priorità.
- Una soluzione generale è quella di aumentare la priorità al processo L portandola ad essere uguale a quella del processo H (solo durante l'attesa di H) in modo che altri processi non possano fare prelazione sul processo L e possa velocemente terminare l'utilizzo della risorsa R (**protocollo di eredità delle priorità**).

Semafori in Java

- Java si possono usare oggetti della classe **Semaphore** (nel package `java.util.concurrent`) che implementa i semafori contatori
- I metodi **acquire()** e **release()** sono gli analoghi di `wait()` e `signal()`
- Il semaforo può essere fair o unfair, se fair le chiamate in attesa su *acquire* vengono accodate con ordine FIFO, se unfair non garantiscono ordinamento (ma sono più veloci)
- Il costruttore vuole due parametri, il valore del semaforo e se fair/unfair, se viene omissso il semaforo è UNFAIR

```
Semaphore mutex=new Semaphore(1,true); //fair
...
mutex.acquire();
count++; //sezione critica
mutex.release();
```

Semaphore

- si possono chiamare **acquire(n)** e **release(n)** anche con parametro intero che indica di quanto decrementare/incrementare il valore del semaforo (indica il numero di 'permessi' che si vogliono acquisire o rilasciare)
- Nel caso di *acquire(n)* aspetterà che il valore del semaforo sia tale da permettere di sottrarre il valore indicato
- E' anche presente metodo *tryAcquire()* che tenta l'acquisizione di un permesso, ritorna true se il permesso è disponibile ed acquisito, ritorna false altrimenti. Attenzione che fa acquisizione unfair, se c'era un thread in attesa non viene data priorità a questo.
- C'è anche metodo *tryAcquire()* con timeout che tenta acquisizione fino ad un tempo massimo
- Nel caso di semaforo unfair è possibile avere attesa indefinita

Semaphore

- schema di uso:

```
Semaphore mutex = new Semaphore(1);  
mutex.acquire();  
try{  
    ... sezione critica ...  
} finally {  
    mutex.release();  
}
```

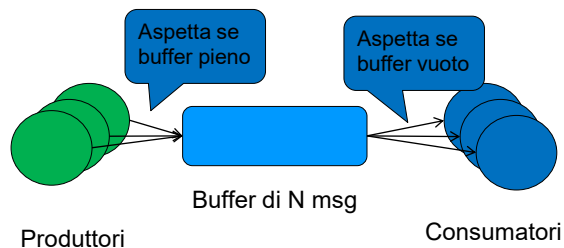
il semaforo rilasciato in
sezione *finally* per
garantire rilascio anche in
caso di eccezione

- Sono disponibili anche classi
 - **ReentrantLock** per gestire mutua esclusione e variabili condizione
 - **ReentrantReadWriteLock** per gestire lock di lettura/scrittura

Classici problemi di sincronizzazione

- Produttori e consumatori con memoria limitata
- Problema dei lettori e scrittori
- Problema dei cinque filosofi

Prod. Cons. con memoria limitata



- buffer di N elementi
- Semaforo **mutex** inizializzato a 1
- Semaforo **piene** inizializzato a 0, indica quante posizioni del buffer sono riempite
- Semaforo **vuote** inizializzato a N, indica quante posizioni del buffer sono vuote

Prod. Cons. con memoria limitata (Cont.)

- Struttura del processo Produttore

```
do {  
    // produce un elemento  
    wait(vuote); // decrementa numero posizioni vuote e aspetta se <= 0  
    wait(mutex); // evita che produttore e consumatore accedano  
                // contemporaneamente al buffer  
    // aggiunge l'elemento al buffer  
    signal(mutex);  
    signal(piene); // incrementa numero posizioni piene ed eventualmente  
                  // sveglia processo consumatore in attesa  
} while (true);
```

Prod. Cons. con memoria limitata (Cont.)

- Struttura del processo Consumatore

```
do {  
    wait(piene); // decrementa n. posiz. piene e attende se <=0  
    wait(mutex); // mutua esclusione con processi produttore  
  
    //rimuove un elemento dal buffer  
  
    signal(mutex);  
    signal(vuote); // incrementa n. posizioni vuote,  
                  // eventualmente sveglia processo in attesa  
    //consuma l'elemento rimosso  
} while (true);
```

Problema dei lettori e scrittori

- Un insieme di dati è condiviso tra processi concorrenti
 - Lettori - leggono i dati e **non** effettuano modifiche
 - Scrittori - leggono e scrivono i dati.
- Problema - permettere a più lettori di leggere contemporaneamente. Solo un singolo scrittore alla volta può accedere ai dati condivisi.
- Dati condivisi
 - I dati
 - Semaforo **mutex** inizializzato a 1.
 - Semaforo **scrittura** inizializzato a 1, indica se la scrittura può essere fatta (1) o no (0)
 - Intero **nLettori** inizializzato a 0, indica il numero di lettori che stanno leggendo

Problema dei lettori e scrittori (Cont.)

- Struttura dei processi Scrittori

```
do {  
    wait(scrittura); //aspetta nel caso scrittura sia 0  
  
    // esegue lettura e scrittura  
  
    signal(scrittura); //porta scrittura a 1, e risveglia  
                      // eventuale processo in attesa  
} while (true)
```

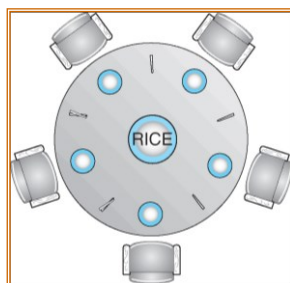
Problema dei lettori e scrittori (Cont.)

- Struttura del processo Lettore

```
do {  
    wait(mutex) ;  
    nLettori ++ ; // incrementa numero processi lettori  
    if (nLettori == 1) //se è il primo lettore  
        wait(scrittura) ; // aspetta nel caso ci sia una scrittura in corso  
                          // altrimenti inibisce la scrittura  
  
    signal(mutex)  
    // esegue lettura  
    wait(mutex) ;  
    nLettori-- ; //decrementa numero processi lettori  
    if (nLettori == 0) // se non ci sono più lettori attivi  
        signal(scrittura) ; // sveglia un eventuale processo scrittore  
    signal(mutex) ;  
} while(true)
```

- Problema: possibile attesa indefinita degli scrittori

Problema dei 5 filosofi



- Dati condivisi
 - Piatto di riso centrale
 - Semafori `bacchetta[5]` inizializzato a 1

Problema dei 5 filosofi (Cont.)

- Struttura filosofo i :

```
do {  
    wait( bacchetta[i] );  
    wait( bacchetta[ (i + 1) % 5] );  
    //mangia  
    signal ( bacchetta[i] );  
    signal ( bacchetta[ (i + 1) % 5] );  
    // pensa  
} while (true) ;
```

- **Problema:** i processi entrano in stallo se tentano tutti di mangiare contemporaneamente

Problemi con i semafori

- Uso non corretto dei semafori:
 - signal (mutex) wait (mutex)
 - Porta ad avere più processi nella sezione critica
 - wait (mutex) ... wait (mutex)
 - Porta allo stallo
 - Omissione di wait(mutex) o signal(mutex) (o entrambi)
 - Porta a stallo o più processi nella sezione critica

Problemi con i semafori

- Uso non corretto dei semafori:

| | |
|-----------------|-------------------|
| wait(mutex) | wait(mutex) |
| ... | ... |
| wait(semaphore) | signal(semaphore) |
| | ... |
| signal(mutex) | signal(mutex) |

- I problemi di sincronizzazione dovuti a errato uso dei semafori sono **molto difficili da individuare** in quanto si possono anche verificare solo in rare occasioni (dipende dai processi in esecuzione, dalla loro velocità) e difficilmente sono ripetibili

Esempi in java

- Produttore consumatore con buffer limitato usando i semafori
- Problema dei lettori-scrittori con semafori
- Cinque filosofi con semafori

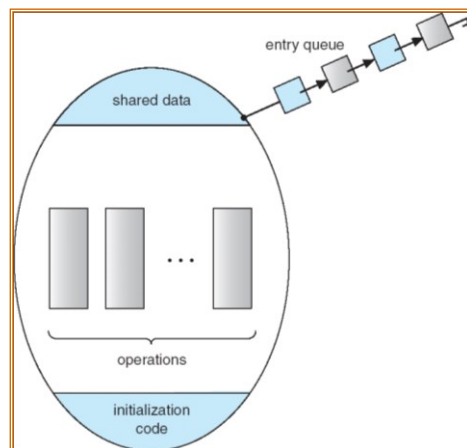
I Monitor

- Una astrazione di alto livello che fornisce un meccanismo conveniente ed efficace per la sincronizzazione dei processi
- **Solo un processo alla volta può essere attivo all'interno di un monitor**

```
monitor nome_monitor
{
    // dichiarazione variabili condivise
    procedure P1 (...) { .... }
    ...
    procedure Pn (...) { ..... }

    initialization code (....) { ... }
    ...
}
```

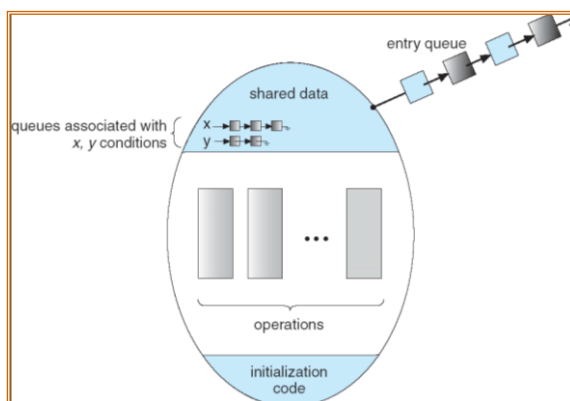
Visione schematica di un Monitor



Variabili condizionali

- `condition x, y;`
- Sono definite due operazioni su variabili condizionali:
 - `x.wait ()` - il processo che invoca l'operazione viene sospeso e implicitamente lascia il monitor e permette ad altri processi di effettuare operazioni sul monitor
 - `x.signal ()` - risveglia uno dei processi che ha invocato `x.wait ()`
- Quando viene eseguita *signal* entrambi i processi potrebbero continuare la loro esecuzione, ma entrambi sarebbero in esecuzione nel monitor, quindi **solo uno dei due può continuare**. E' scelta dell'implementazione decidere a chi dare l'accesso dopo la signal

Schema di un Monitor con variabili condizionali



Soluzione dei 5 filosofi

```
monitor cinque_filosofi
{
    enum { PENSA, AFFAMATO, MANGIA } stato [5];
    condition self [5];

    void prendi(int i) {
        stato[i] = AFFAMATO;
        verifica(i);
        if (stato[i] != MANGIA) self[i].wait();
    }

    void lascia(int i) {
        stato[i] = PENSA;
        // controlla i filosofi vicini a destra e sinistra
        verifica((i + 4) % 5);
        verifica((i + 1) % 5);
    }
}
```

Soluzione dei 5 filosofi (cont)

```
void verifica (int i) {
    if ( (stato[(i + 4) % 5] != MANGIA) &&
        (stato[i] == AFFAMATO) &&
        (stato[(i + 1) % 5] != MANGIA) ) {
        stato[i] = MANGIA;
        self[i].signal();
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
        stato[i] = PENSA;
}
}
```

Soluzione dei 5 filosofi (cont)

```
monitor cinque_filosofi cf;

process filosofo(i)
{
    do {
        // pensa
        cf.prendi(i);
        // mangia
        cf.lascia(i);
    } while (true);
}
```

Monitor

- Il costrutto monitor (o simile) è presente in alcuni linguaggi come il concurrent pascal, mesa, C# e Java
- Un monitor può essere realizzato usando i semafori

Implementazione Monitor con semafori

- Un mutex viene usato per garantire accesso ad un solo processo al monitor, ogni procedura F del monitor trasformata in:

```
wait(mutex);
...
body of F
...
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

- il semaforo next ed il contatore next_count (inizializzati a zero) servono a garantire l'accesso al monitor ad un solo processo nel caso di una signal su una condizione (vedi dopo)

Implementazione Monitor con semafori

- Per ogni variabile condizione x viene usato un semaforo x_sem e un contatore x_count (inizializzati a zero)

Wait:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

x.wait()

```
// incrementa n. di thread in attesa di x
// se c'è un thread che ha fatto signal, in attesa monitor
// lo sveglia
// altrimenti
// sblocca il mutex
// aspetta x
// decrementa n. thread in attesa di x
```

Signal:

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

x.signal()

```
// se c'è un thread in attesa di x
// incrementa thread in attesa di entrare nel monitor
// segnala a un thread in attesa di x
// aspetta di rientrare nel monitor
// decrementa thread in attesa di entrare nel monitor
```

Monitor in Java

- Java implementa i monitor potendo definire i metodi **synchronized**, viene associato all'oggetto (o alla classe se metodo statico) un mutex che permette solo ad un metodo synchronized l'esecuzione del suo codice, il mutex sarà rilasciato quando il metodo termina (anche in caso di eccezione)

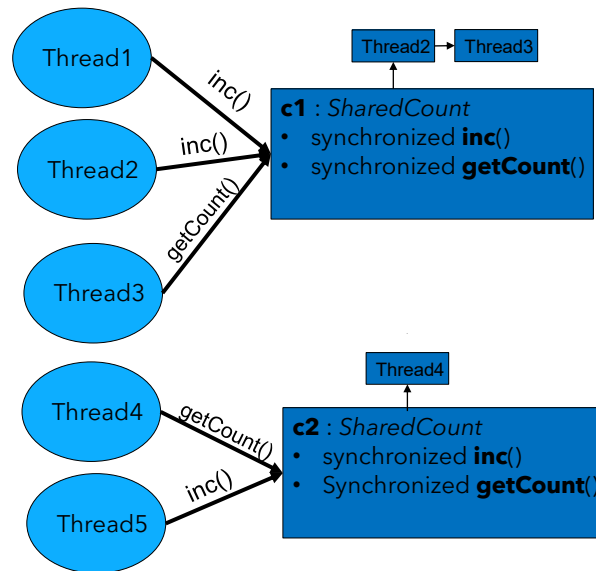
```
class SharedCount {  
    private int count = 0;  
    public synchronized void inc() {  
        count++;  
    }  
    public synchronized int getCount() {  
        return count;  
    }  
}
```

Se due o più thread usano lo stesso oggetto SharedCount ed entrambi chiamano il metodo *inc()* o *getCount()* solo uno alla volta possono incrementare count o accedere al valore di count

metodi synchronized

- Inoltre se dal codice di un metodo synchronized viene chiamato un altro metodo synchronized il mutex non viene richiesto nuovamente (entrerebbe in stallo) ma il mutex verrà rilasciato solo alla terminazione del metodo iniziale (viene detto lock rientrante)
- I metodi **statici** synchronized usano un mutex associato alla classe

Monitor in Java



Wait e Notify

- in un metodo sincronizzato è possibile usare i metodi **wait** e **notify** in modo simile a *wait* e *signal* nelle condizioni di un monitor
- Il metodo **wait()** rilascia il mutex associato al metodo **synchronized** e mette il thread in attesa che venga chiamato il metodo **notify/notifyAll**
- il metodo **notify()** estrae un thread in attesa sull'oggetto, se presente, che viene riavviato e messo in attesa di poter acquisire il mutex sull'oggetto (altrimenti si avrebbero due thread in esecuzione su due metodi **synchronized** dello stesso oggetto) quando lo acquisirà continuerà ad eseguire il codice dopo invocazione metodo **wait()**
- il metodo **notifyAll()** è analogo a **notify()** solo che vengono tolti dalla coda e risvegliati tutti i thread in attesa sull'oggetto.
- nota: **wait()**, **notify()** e **notifyAll()** si possono chiamare solo da dentro un metodo **synchronized**.

Wait e Notify

- Il metodo `wait` per attendere una specifica condizione va usato nel seguente modo:

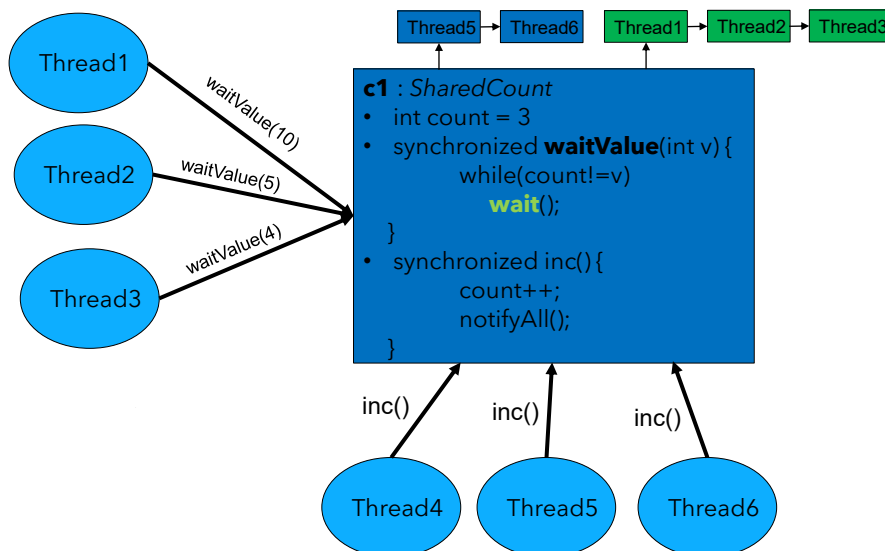
```
...  
while(! condizione)  
    wait();  
...
```

- mentre quando la condizione diventa vera si deve fare:

```
...  
condizione = true;  
notifyAll();  
...
```

- In genere è più sicuro usare `notifyAll()`, per esempio quando si possono avere su un oggetto più thread in attesa su condizioni di tipo diverso. Usando `notify()` c'è rischio di non risvegliare il thread che è in attesa sulla condizione diventata vera (che quindi viene persa). Ma risvegliare tutti i thread in attesa (potrebbero essere molti) ha un impatto sulla performance perché tutti i thread rientreranno in esecuzione e chiederanno di entrare nel monitor per valutare la condizione

Wait e Notify



istruzione synchronized

- E' possibile anche sincronizzare non un metodo intero ma un blocco di istruzioni sulla base di un oggetto qualsiasi. La sintassi è:

```
...
synchronized(object) {
    istruzioni...
}
...
```

- Le istruzioni sono eseguite solo dopo che è stato acquisito il mutex sull'oggetto *object* il mutex verrà rilasciato all'uscita dal blocco, nelle istruzioni si può eseguire wait e notify/notifyAll su *object*
- Permette di sincronizzare solo le istruzioni che possono generare una race condition
- Un metodo synchronized m() è equivalente a:

```
void m() {
    synchronized(this) {
        ...istruzioni del metodo...
    }
}
```

Esempi di monitor in java

- Semaforo contatore usando metodi synchronized, wait e notify
- Produttore-consumatore con buffer limitato
- Lettori-scrittori usando synchronized
- Filosofi a cena senza deadlock