

PROBLEMA DEI LETTORI-SCRITTORI > 7.1.2

È uno dei principali problemi di sync. Definisce una base di dati alla quale N processi divisi tra lettori e scrittori vi vogliono accedere in modo concorrente.

PROBLEMI DEL LS

Sono variabili, che implicano l'esistenza di priorità.

• 1° Problema

nessun lettore deve attendere, tranne quando uno scrittore ha già l'accesso.

Soluzioni di questo problema possono portare a starvation degli scrittori

• 2° Problema

Se uno scrittore è pronto, allora i lettori devono attendere.

Può portare a starvation dei lettori.

IMPLEMENTAZIONE 1° PROBLEMA

Si definiscono 2 semafori + 1 contatore:

- rw-mutex : Semaforo di mutua esclusione per
 - tutti gli scrittori
 - primo o ultimo lettore che entra o escano dalla sezione critica.
- mutex : Semaforo di mutua esclusione usato per l'aggiornamento di read-count
- read-count : Conta i processi che stanno leggendo.

SCRITTORE

Aspetta la m.esclusione di rw-*.
poi scrive e libera rw-*

```
while (true) {
    wait(rw_mutex);
    ...
    /* esegui l'operazione di scrittura */
    ...
    signal(rw_mutex)
}
```

Figura 7.3 Struttura generale di un processo scrittore.

LETTORE

1. Aspetta mutex per incrementare count
2. Se count è 1 (lui è il 1°), Aspetta rw-* per aprire mutua esclusione
3. Rilascia mutex e legge
4. Aspetta mutex per decrementare count
5. Se count=0 lera l'ultimo rilascia rw-* per fare lettori e Rilascia mutex

```

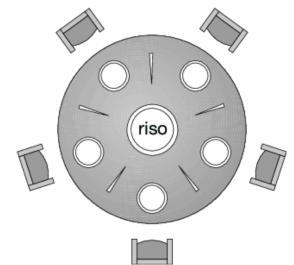
while (true) {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    ...
    /* esegui l'operazione di lettura */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
}

```

Sono presenti soluzioni generalizzate che danno a N processi il lock in lettura, ma a 1 lock in scrittura per la mutua esclusione.

PROBLEMA DEI 5 FILOSOFI ➤ 7.1.3

Describe 5 filosofi seduti su una tavola rotonda. Sul tavolo ci sono 5 bacchette ma per mangiare ne servono 2.



Ogni filosofo o pensa (posa bacchette) o mangia.

Questo meccanismo crea dei problemi quando tutti provano a mangiare.

SOLUZIONE CON SEMAFORI 7.1.3.1

Si definisce un semaforo per ogni bacchetta.

Dopo che ogni i-esimo filosofo:

1. Prende le bacchette a sx : `wait(chopstick[i])`
2. Prende // // a dx : `wait(chopstick[i+1 % 5])`
il %5 è per prendere la 1 se è l'ultimo
3. Mangia e in maniera speculare fa `signal`.

Questa soluzione non è sufficiente perché vulnerabile a stallo se tutti i filosofi hanno fame e afferrano la sx.

Si possono comunque risolvere applicando dei limiti:

- solo 4 filosofi
- check bacchette sx e dx
- Prima i pari, poi i dispari

```

while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    ...
    /* mangia */
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    /* pensa */
    ...
}

```

Figura 7.6 Struttura del filosofo i.

SOLUZIONE CON MONITOR 7.1.3.2

La soluzione applica il vincolo che un filosofo possa mangiare solo se ha entrambe le bacchette libere.

STRUTTURE DATI

Si definiscono

- stati filosofo: Thinking, Hungry, Eating

Si può impostare Eating solo se i suoi vicini non stanno già mangiando

- condizione :

Si scandisce soc e dx in maniera circolare

- condition variables:

Si definiscono per mettere in attesa i processi che hanno fame ma non riescono a mangiare

condition self[5];

FUNZIONE DI TEST

Verifica che il filosofo i possa mangiare e se si, gli imposta stato e fa la signal della cond variable

```

void test(int i){
    enum state_left = state[i + 4] % 5;
    enum state_right = state[i + 1] % 5;
    enum i_state = state[i];
    if (
        state_left != EATING // left is not eating
        && i_state == HUNGRY // i'm hungry
        && state_right != EATING // right is not eating
    ){
        state[i] = EATING; // setting self to EATING
        self[i].signal(); // freeing self from cond. variable
    }
}

```

CODICE DEL MONITOR

Definisce, oltre alle cose già dette, 2 funzioni

- pickup(i): prova a prendere le bacchette, mettendosi in attesa se non ci riesce.
- putdown(i): "Posa le bacchette" re-impostando lo stato e chiamando il test sui vicini

Con questa soluzione, può succedere che un filosofo attende indefinitivamente.

```

...
monitor DiningPhilosophers{
    // data section
    enum { THINKING, HUNGRY, EATING } state[5];
    condition self[5]; // cond. variables

    // function section
    void pickup(int i){
        state[i] = HUNGRY; // sets as hungry
        test(i); // tries to set as EATING
        if (state[i] != EATING){
            self[i].wait();
        }
    }

    void putdown(int i){
        state[i] = THINKING;
        // calls the test to the near elements
        int left = (i+4) % 5;
        int right = (i+1) % 5;
        test(left);
        test(right);
    }

    void test(int){ /*...*/}
}

```

SINCRONIZZAZIONE NEL KERNEL > 7.2

Gli S.O. windows e linux hanno diverse gestioni di sync.

Sync in windows 7.2.1

Datta una risorsa globale **dentro il kernel**:

- sistemi monoprocessoressi:

maschera gli interrupt che potrebbero modificare la risorsa.

- sistemi multiprocessoressi:

usa semafori ad attesa attiva (spinlock) e impedisce le prelazioni dei thread con spinlock.

mentre fuori dal kernel si vedono i dispatcher che possono **prelevarla** essere signaled (risorsa disp) e unsigned (risorsa bloccata).

Sync in linux 7.2.2

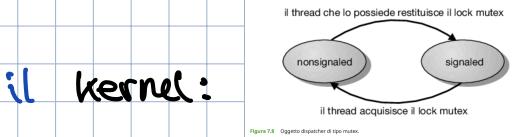
per le operazioni matematiche

si possono usare gli interi atomici
che non permettono interruzioni.

Mette anche a disposizione mutex_lock(); e mutex_unlock();

Infine a seconda che sia mono o multi processore

modificherai la prelazione / spinlock



```
1 atomic_t counter;
2 int value;
3
4 // operations
5 atomic_set(&counter, 5);           // counter = 5
6 atomic_add(10, &counter);          // counter = counter + 10
7 atomic_sub(4, &counter);           // counter = counter - 4
8 atomic_inc(&counter);             // counter++
9 value = atomic_read(&counter);    // value = 12
```

monoprocessoressi

multiprocessoressi

Inibisce la prelazione a livello kernel. Attiva spinlock.

monoprocessoressi

multiprocessoressi

Abilita la prelazione a livello kernel.

Rimuove spinlock.

POSIX sync > 7.3 Area non coperto a lezione

Per la sincronizzazione di programmi esterni al kernel

POSIX mette a disposizione varie funzionalità:

LOCK MUTEX <pthread.h>

Viene messo a disposizione il tipo pthread_mutex_t insieme a dei metodi di gestione

SEMAFORI <semaphore.h>

I semafori sono fuori dello standard POSIX.

Possono essere

```
1 #include <pthread.h>
2
3 pthread_mutex_t mutex;
4
5 // initializes mutex
6 pthread_mutex_init(&mutex, NULL);
7
8 // acquires lock
9 pthread_mutex_lock(&mutex);
10
11 // critical section
12
13 // releases lock
14 pthread_mutex_unlock(&mutex);
```

- named: Hanno un nome impostato dall'esterno, utile per proc. diff.
- un named: Non hanno nome e possono essere usati solo dai thread dello stesso proc.

VARIABILI CONDIZIONALI

Si realizzano in concordanza di un lock mutex col tipo `pthread_cond_t`

```
1 // defining variables
2 pthread_mutex_t mutex;
3 pthread_cond_t cond_var;
4
5 // initializing condition variable
6 pthread_mutex_init(&mutex, NULL);
7 pthread_cond_init(&cond_var, NULL);
```

```
1 // locking to protect the check from
2 // race condition
3 pthread_mutex_lock(&mutex);
4
5 // verifying condition a==b
6 while(a != b){
7     // if not, then it will release mutex
8     // and give space to other threads
9     pthread_cond_wait(&cond_var, &mutex);
10 }
11
12 pthread_mutex_unlock(&mutex);
```

```
1 // setting condition a=b
2 pthread_mutex_lock(&mutex);
3
4 a=b
5
6 // signaling the condition variable
7 pthread_cond_signal(&cond_var);
8
9 pthread_mutex_unlock(&mutex);
10 // from here, the other process will continue from the while
11 // loop
```

Inizializzazione. Aspetto condizione Imposta condizione.

APPROCCI ALTERNATIVI > 7.5

Approcci alternativi per la sincronizzazione sono.

Memoria Transazionale

Sfrutta il concetto dei database per creare delle transactions per fare le modifiche dei dati conclusi.

OPENMP

si può usare la direttiva col preprossore `#pragma omp critical`.

STALLO DEI PROCESSI > 8

Il deadlock / stallo si verifica quando processi si aspettano a vicenda.

DEFINIZIONE MODELLO > 8.1

Un sistema si può rappresentare come un insieme di risorse che vengono distribuite su thread in competizione. Le risorse sono

- suddivisibili per tipo es: CPU, IO....
- formate da istanze identiche es: CPU₁, CPU₂,...

```
#include <semaphore.h>

sem_t *n_sem;

5 // creates a named semaphore
6 // "SEM" => Name
7 // 0_CREAT => If not exists, create the semaphore
8 // 0666 => Specifies access to other processes
9 // 1 => Initial value
10 n_sem = sem_open("SEM", 0_CREAT, 0666, 1);
11
12 // Equivalent to wait()
13 sem_wait(n_sem);
14
15 // Equivalent to signal()
16 sem_post(n_sem);
17
18 sem_t *u_sem;
19
20 // Creates an unnamed semaphore
21 // 0 => specifies that the semaphore can only be used in
22 // the threads of the parent process
23 // 1 => Initial value
24 sem_init(&u_sem, 0, 1);
25
26 // Equivalent to wait()
27 sem_wait(&u_sem);
28
29 // Equivalent to signal()
30 sem_post(&u_sem);
31
32
33
```

Uso delle risorse

Le risorse vengono usate secondo lo schema

- request:

Il thread richiede l'accesso alla risorsa e si mette in attesa se non è imm. disponibile.

- use: Il thread opera sulla risorsa.

- release: Il thread rilascia la risorsa.

A livello pratico si usano le system call a seconda della risorsa es: malloc(); e free(); per memoria, open(); close(); per file ecc.. L'S.O. detiene una tabella delle risorse con eventuale thread assegnato.

CARATTERIZZAZIONE STALLO ➤ 8.3.1

Lo stallo si verifica solo se sono presenti:

- Mutual exclusion:

Esiste almeno 1 risorsa non condivisibile.

- Hold and wait:

Il thread entra in possesso di 1 prima risorsa e aspetta di ottenerne altre (che potrebbero essere in possesso di altri thread).

- No preemption:

Ogni risorsa occupata può essere rilasciata solo volontariamente dal thread.

- Circular wait:

Esiste un gruppo di thread $\{T_0, T_1, \dots, T_n\}$ in cui ogni thread aspetta delle risorse in possesso del succ.

Es: $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t_0$

RESOURCE ALLOCATION GRAPH ➤

È una rappresentazione che aiuta a capire se un sistema presenta o meno dei dead lock.

SINASSI

Si definiscono una serie di vertici V e archi E.

V può essere diviso in

- $T = \{t_1, t_2, \dots, t_n\}$ insieme dei thread.
- $R = \{r_1, r_2, \dots, r_n\}$ insieme delle risorse.

] Di sistema

Le relazioni E possono quindi essere

- request edge: $T_i \rightarrow R_j \quad \forall i, j$
rappresenta la richiesta del thread i di ottenere la risorsa j .
- assignment edge: $R_j \rightarrow T_i \quad \forall i, j$
rappresenta l'assegnazione della risorsa j al thread i .

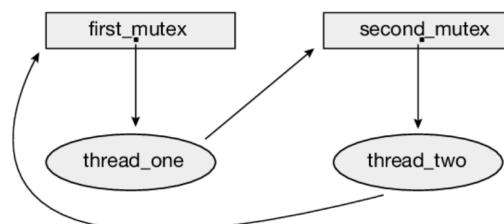
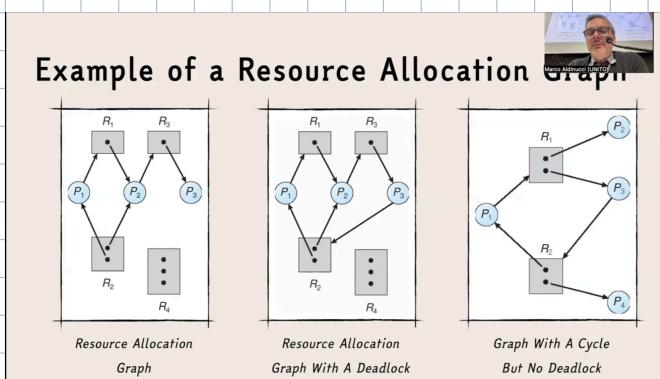


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.



Può plus rompere lo stato

Per rapp. le istanze, si mettono dei quadratini.

I **■** devono essere specificati in assegnazione.

I **deadlock** si verificano in base ai cicli:

- 0 cicli: no deadlock
- >0 cicli:
 - $\forall r$ ha 1 istanza: **deadlock**
 - $\forall r$ ha >1 ist: dipende