

# SOLUZIONE DI PETERSON > 6.3

È una prima soluzione al problema della critical selection con l'assunzione che non ci siano load e store parziali.

## FUNZIONAMENTO

Per i 2 processi  $P_i$  e  $P_j$ , condividono:

```
int turn;  
  
boolean flag[2];  
  
while (true) {  
  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
  
    /* sezione critica */  
    flag[i] = false;  
  
    /* sezione non critica */  
}  
  
Figura 6.3 Struttura del processo  $P_j$  nella soluzione di Peterson.
```

- turn: indica il turno di accesso alla sezione critica

- Flag: indica se un processo è pronto a entrare nella sezione critica

Se  $P_i$  vuole l'accesso:

1.  $\text{flag}[i] = \text{true}$  indica che è pronto
2.  $\text{turn} = j$ : conferisce all'altro processo l'accesso  $\rightarrow$  per garantire i principi

## DIMOSTRAZIONE

- Mutua esclusione

Applicando de Morgan sulla guardia del while,  $P_i$  può entrare se  $\neg \text{flag}[j] \vee \text{turn} = i$  e che se entrambi potessero entrare allora  $\text{flag}[i] = \text{flag}[j] = \text{true}$  che contraddice la guardia.

- Progresso / Attesa limitata

$P_i$  e  $P_j$  si impostano a vicenda i turni e si de-selezionano appena hanno finito, garantendo le prop.

## PROBLEMI CON ARCHITETTURE RISERVE

Nelle architetture multi-core, è probabile che load/store avvengano in ordine diverso rispetto a quanto dichiarato nel codice:

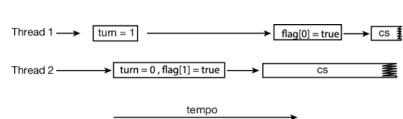


Figura 6.4 Effetti del riconfigurazione delle istruzioni nella soluzione di Peterson.

Questo significa che non può funzionare perché non è garantito l'ordine. Occorre un mech. di sync

## HARDWARE SYNC > 6.4.2

Per le architetture moderne occorre realizzare una sincronizzazione che coinvolge anche S.O. / HW.

Le soluzioni si basano su un meccanismo di lock che esclusiva la sezione critica da interferenze.

- Uni processors : Impedimento degli interrupt
- Atomic instructions: Operazioni indivisibili

### TEST AND SET

È un istruzione atomica che, dato un bool, restituisce il suo valore corrente e nel mentre assegna al parametro true.  
Diventa atomica perché codificata come unica istruzione assembly.

### IMPLEMENTAZIONE LOCK

Si usa una variabile condivisa **lock-false**

Dopodiché

- Il while va avanti fin quando **lock = true** ovvero è già presente il lock
- Uscito dal ciclo, **test\_and\_set()**, torna **false** perché il lock è libero, ma assegna **lock = true** per dire che il processo corrente è in lock.
- Terminata la sezione critica, **lock=false** per segnalare la fine della sezione critica.

Il test-and-set garantisce mutua esclusione e progresso, ma ci si affida all'hw per garantire l'attesa limitata.

### COMPARE AND SWAP

Simile a test and set. Restituisce sempre il val. originale di value.

Se **value = expected** allora imposta new value.

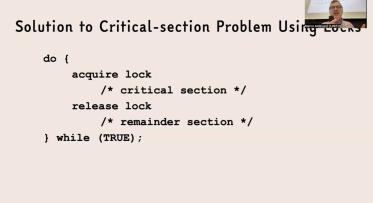


Figura 6.5 Definizione dell'istruzione atomica test\_and\_set().

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

Figura 6.6 Realizzazione di mutua esclusione con test\_and\_set().

```
do {
    while (test_and_set(&lock));
    /* non fa niente */

    /* sezione critica */

    lock = false;

    /* sezione non critica */
} while (true);
```

Figura 6.7 Definizione dell'istruzione atomica compare\_and\_swap().

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

Il vantaggio del CAS è che si possono usare word arbitrarie al posto di bool lock - o

## IMPLEMENTAZIONE LOCK

Il funzionamento generale è identico a test and set.

Nella versione con attesa limitata

- Key viene usato come il vecchio val di lock.  
→ Iniz. a false
- waiting[i] definisce per ogni processo, quelli che aspettano e quello in esecuzione (solo 1)
- Nel 1° while, Pi rimane in attesa fin quando c'è il lock ed è dichiarato in attesa. Ad ogni iterazione il CAS assicura il valore corretto del lock
- Appena uscito dal while,  $\neg \text{waiting}[i] \vee \neg \text{key}$ , quindi il processo può entrare nella sezione critica.
- Dopo di che viene scandito waiting[i] in maniera circolare per trovare il 1° elemento in attesa
  - o se si torna a se stessi:
    - Se si torna a se stessi: libera il lock
    - Se si trova un proc. in attesa:  
si mette se stessi non più in attesa ma non si tocca il lock perché è già occupato

Questo meccanismo garantisce l'attesa limitata perché ogni processo aspetterà al massimo n-1 torri.

Le 2 metodologie sono già implementate con pthreads.

## SEMAFORI > 6.6

È un meccanismo di sincronizzazione tra n elementi comunicanti. Solitamente è usato nel modello shared mem.

## FUNZIONAMENTO

```
while (true) {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* non fa niente */  
  
    /* sezione critica */  
  
    lock = 0;  
  
    /* sezione non critica */  
}
```

Figura 6.8 Realizzazione di mutua esclusione con compare\_and\_swap().

```
while (true) {  
    waiting[i] = true;  
    key = 1;  
    while (waiting[i] && key == 1)  
        key = compare_and_swap(&lock, 0, 1);  
    waiting[i] = false;  
  
    /* sezione critica */  
  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
  
    if (j == i)  
        lock = 0;  
    else  
        waiting[j] = false;  
  
    /* sezione non critica */  
}
```

Figura 6.9 Mutua esclusione con attesa limitata con compare\_and\_swap().

Si usa una variabile S intesa nella quale si può accedere (escudendo inizializzazione) solo con:

- wait(S); equivalente di lock();
- signal(S); equivalente di unlock();

La variabile rappresenta il # di "posti liberi" che viene incrementato e decrementato da create() e signal().

wait(), e signal(); sono atomiche (se  $S=1$ , è = (lock(); unlock)).

## Uso del semaforo

A differenza delle lock che vengono usate principalmente per accessi concorrenti alla stessa risorsa, i semafori sono pensati per garantire ordine di esecuzione tra processi.

Esempio →

## Attesa attiva e soluzione

Quando un processo è in attesa con wait(); dal punto di vista del S.O. sarà in CPU burst per tutto il tempo, comportando spreco di risorse. (busy wait)

### Idea di soluzione

Per risolvere il problema, il processo si può fermare da solo passando in stato wait, per poi essere risvegliato quando viene invocata signal(); per passare in stato ready.

### IMPLEMENTAZIONE

Sarà il semaforo ad implementare una struttura per definire i processi in attesa.

Il valore del semaforo (se negativo) indica il # dei processi in cattiva attesa.

## 6.6 Semaphore

Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.  
• Semaforo S - integer variable  
• Can only be accessed via two indivisibile (atomic) operations  
wait() and signal()  
Originally called P() and V()  
• Definition of the wait() operation  
wait() {  
 while (S < 0)  
 ; // busy wait  
 S--;  
}  
• Definition of the signal() operation  
signal() {  
 S++;  
}

(se  $S=1$ , è = (lock(); unlock)).

### Semaphore Usage

• Counting semaphore - integer value can range over an unrestricted domain  
• Binary semaphore - integer value can range only between 0 and 1  
• Same as a mutex lock  
• Can solve various synchronization problems  
• Consider P1 and P2 that require S1 to happen before S2

• Create a semaphore "synch" initialized to 0  
P1:  
 s1:  
 signal(synch);  
 P2:  
 wait(synch);  
 S2;  
• Can implement a counting semaphore as a binary semaphore

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;  
  
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```

## AUTOCIA OPERAZIONI

Le operazioni di `wait()` e `signal()` devono essere atomiche ovvero bisogna evitare che 2 processi possano chiamarli simultaneamente:

- 1 CPU: Basta imbire gli interrupt.

- 2+ CPU: Bisogna implementare meccanismi di lock. → CAS

## DEADLOCK & STARVATION > 6.8

Sono situazioni anomale che si presentano sia con i lock che con i semafori.

### DEADLOCK

Si presenta quando 2 processi si attendono l'uno l'altro.

Una possibile soluzione sarebbe

di stabilire un ordine di esecuzione

- Deadlock - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let S and Q be two semaphores initialized to 1

```
P0          P1
wait(S);   wait(Q);
wait(Q);   wait(S);
...
signal(S); signal(Q);
signal(Q); signal(S);
```

### STARVATION

Si verifica quando un processo rimane in attesa indefinita-