Sezione critiche

Una stessa struttura dati viene utilizzata da due thread contemporaneamente (race condition). Una sezione critica è una porzione di codice che , con riferimento ad una data struttura dati, può essere eseguita da un solo thread alla volta.

Una possibile soluzione sta nel garantire la mutua esclusione di accesso alle risorse , ovvero in un determinato istante di tempo solo un thread deve poter accedere alla risorsa. La soluzione deve essere applicabile in diversi contesti e non deve essere condizionata da fattori variabili da un sistema all'altro, ad esempio timing, ipotesi sul numero di CPU, ecc..

Nessun processo al di fuori della sezione critica deve bloccare logicamente un altro thread che vuole accedere alla propria sezione critica. Quando un processo deve entrare in sincronizzazione con un altro necessita di essere bloccato ma vogliamo che i thread vengano bloccati il minimo indispensabile.

Nessun processo thread deve rimanere in attesa all'infinito in attesa di entrare nella propria sezione critica : dobbiamo poter garantire un tempo massimo di attesa ad un processo.

Il flusso di esecuzione di due thread A e B concorrenti deve avvenire in modo che se il thread A entra nella propria zona critica , allora il thread B entra in uno stato logico di "bloccato" a seguito di un sistema di sincronizzazione nell'arco di tempo che il thread A lavora in sezione critica. Quando A termina la propria zona esecuzione in una zona critica, allora B può essere sbloccato. Si nota che è importante ridurre al minimo il tempo di blocco.
A causa di prelazione è basato su un timer che periodicamente fa scattare una IRQ che blocca il processo in esecuzione, per esempio a seguito di multiplexing temporale.

Si può pensare di disabilitare i meccanismi di gestione delle interrupt prima della sezione critica e riattivarla dopo la fine dell'esecuzione , ma questa soluzione per garantire la mutua esclusione funziona solo nei processori con una sola CPU perchè le notifiche di prelazione, nonostante siano bloccate in una CPU, continuano ad arrivare nelle altre CPU e se il thread concorrente va ad accedere alla stessa risorsa ma da una CPU differente. Questa soluzione quindi non funziona sui sistemi multicore, quindi non è una soluzione accettabile. Quando vengono disabilitati i meccanismi di gestione delle Interrupt , bisogna riattivarli il prima possibile in modo da ripristinare il corretto funzionamento del SO. Inoltre un thread (processo utente) dovrebbe disattivare questi meccanismi , ma quest'azione può essere eseguita solo in kernel mode. Questa soluzione di fatto viene utilizzata all'interno del sistema operativo ed in modo responsabile e solo sui sistemi a singolo Core.

Un meccanismo differente potrebbe essere quello di pensare ad una variabile di LOCK condivisa che normalmente è a 0 , ovvero nessun thread è bloccato mentre quando la variabile vale 1 allora si ha il blocco di un thread. Quando si entra in una regione critica si chiama la funzione "enter_region()" e quando si lascia la zona critica "leave_region()". Intuitivamente queste funzioni dovrebbero prendere la variabile di lock L . Se la enter_region() trova la variabile L a 0, la pone a 1 e prosegue l'esecuzione mentre se la trova già ad 1 allora non deve permettere l'esecuzione del thread. Un meccanismo di busy waiting legge ciclicamente la variabile rendendo inefficiente il funzionamento del meccanismo ma si presenta un'ulteriore problema : due thread potrebbero leggere la variabile contemporaneamente vedendola a 0 e quindi porla a 1 , ma questo si traduce in un'istanza del problema di partenza e quindi è inutile.

Una soluzione valida è invece quella dell'**Alternanza Stretta** : il meccanismo assicura che prima tocca ad un processo di entrare in regione critica e poi ad un altro. L'enter_region() a questo punto prende in input l'identificativo del thread chiamante, se l'identificativo coincide col turno in atto , descritto dalla variabile TURN, allora il processo prosegue con l'esecuzione mentre la leave_region() si preoccupa di "lasciare la porta aperta" cambiando il turno , in modo che l'altro processo possa entrare nella enter_region(). Il meccanismo di alternanza stretta garantisce la mutua esclusione

```
int N=2
int turn

function enter_region(int process)
   while (turn != process) do
      nothing

function leave_region(int process)
   turn = 1 - process
```

Questa soluzione si adatta all'esistenza di N processi grazie all'imposizione di turni rigidi.
Un processo però, al di fuori della propria sezione critica, può bloccare un processo che si trova nella propria sezione critica (viola il principio 3).

Si può ovviare al problema grazie alla **soluzione di Peterson**, una soluzione software che non fa uso di harware(quindi universale). Si usano ancora delle variabili condivise :
Si hanno solo due processi (al momento non adattabile al caso di N processi). Con un vettore Interested (inizialmente tutto a false) un processo segnala la propria volontà di essere interessato ad entrare in regione critica. Si effettuano degli aggiornamenti nelle uniche due variabili condivise. Ogni processo è proprietario del proprio slot, ovvero il processo 1 è l'unico che può modificare il proprio slot dell'array interested, e potrebbe al più leggere gli altri slot durante la enter_region() . Un processo entrerà in uno stato di blocco se il turno in atto coincide con l'identificativo dell'altro processo ed inoltre quello stesso processo è interessato ad entrare in zona critica. Questo risolve il problema che un processo non interessato riesca a bloccare un processo che vorrebbe entrare in sezione critica.

1. Inizialmente, entrambi i processi A e B impostano il proprio flag su "falso".
2. Quando il processo A vuole accedere alla risorsa condivisa, imposta il proprio flag su "vero".
3. Il processo A imposta la variabile di turno al valore di "processo B".
4. Il processo A aspetta finché il "processo B" non ha finito di accedere alla risorsa condivisa (cioè il flag del "processo B" è "falso" o il turno è quello del "processo A").
5. Quando il "processo B" ha finito di accedere alla risorsa condivisa, imposta il proprio flag su "falso", lasciando l'accesso alla risorsa al "processo A".
6. Il "processo A" può ora accedere alla risorsa condivisa, seguendo gli stessi passaggi.

Quando alla variabile del turno vogliono accedere contemporaneamente due thread interessati la variabile verrà scritta da chi arriva per primo e sovrascritta dal thread che "perde". La seconda condizione del while serve proprio a bloccare l'esecuzione del thread corrente nel caso in cui il turno sarà diverso dall'id del processo, perchè chi ha fatto per secondo lo "store" nel turno , avrà impostato il flag e vede turn = process.

Verrà sovrascritta dal thread che arriverà secondo

```
int N=2
int turn
int interested[N]

function enter_region(int process)
    other = 1 - process
    interested[process] = true
    turn = process
    while (interested[other] = true and turn = process) do
        nothing

function leave_region(int process)
    interested[process] = false
```

Questa soluzione non è valida per i sistemi multiprocessore in quanto un meccanismo di pipelining riodini delle istruzioni commutabili a cui si avrà accesso da altri core e le letture avverranno in modo inconsistente. Le variabili interested[process] ed interested[other] possono essere lette entrambe a false, quando invece erano state impostate a true , a causa di un riordino di istruzioni dettato dal pipeline.

Istruzione TSL o (XCHG) **TSL R , [L]**

Una soluzione che fa uso di un supporto speciale dell'hardware è invece quella di fare uno dell'istruzione TSL (test and set lock) mentre su CPU IntelX86 si ha l'equivalente XCHG.

L'istruzione TSL viene utilizzata per garantire che solo un processo alla volta possa accedere ad una risorsa condivisa, acquisendo un lock sulla risorsa stessa. L'istruzione TSL esegue due operazioni: verifica se il lock è disponibile e lo acquisisce se lo è. Se il lock non è disponibile, il processo che esegue l'istruzione TSL viene sospeso fino a quando il lock non diventa disponibile.

L'istruzione XCHG, invece, viene utilizzata per scambiare il contenuto di due registri o di due locazioni di memoria atomicamente. L'istruzione XCHG esegue tre operazioni: salva il contenuto di un registro in una variabile temporanea, copia il contenuto di un altra locazione di memoria nel primo registro, e infine scrive il contenuto della variabile temporanea nel secondo registro (swap classico).

L'atomicità di questa operazione rende la soluzione dell'alternanza stretta funzionante. Non essendo due istruzioni divise essa non potrà interlacciarsi con altre istruzioni. In oltre la TSL blocca il buffer di esecuzione delle istruzioni in modo da evitare che , in una struttura multicore, non si abbia il problema di accesso simultaneo. Abbiamo un modo atomico di eseguire fetch and store anche su sitemi multicore. Si usa la TSL per implementare la enter_region(). La TSL si usa per testare se la variabile di Lock è a 0 e settarla a 1. Dopo l'esecuzione della istruzione TSL la Lock vale sicuramente 1, si guarda il valore del registro della TSL per valutare se era 0 anche prima dell'esecuzione della TSL , in quel caso entro nella regione critica. La leave_region() imposta il valore della Lock a zero. Questa è una soluzione che funziona su architetture multicore e che sfrutta istruzioni in modalità utente.

Una variabile di Lock è sempre associata ad una determinata struttura dati per gestirne l'accesso concorrente da parte di differenti thread.

Inversione di priorità

Rimane ancora il problema del busy waiting con l'uso di istruzioni TSL. Si ha un problema di inversione di priorità (un processo ad alta priorità è bloccato da un processo a bassa priorità) che causa lo stallo dei processi.

L'inversione di priorità può generare un loop attraverso lo spin lock in una situazione in cui un processo ad alta priorità è bloccato in attesa di una risorsa detenuta da un processo a bassa priorità che è stato sospeso dallo scheduler del sistema operativo, ma non ha rilasciato la risorsa.

In questa situazione, il processo ad alta priorità continuerà a verificare ripetutamente la disponibilità della risorsa (cioè continuerà a girare in loop attraverso lo spin lock) perché il processo a bassa priorità è stato sospeso e non è in grado di rilasciare la risorsa. Questo loop attraverso lo spin lock può causare un'elevata utilizzazione della CPU e rallentare il sistema operativo, poiché il processo ad alta priorità continua a cercare la risorsa senza successo.

Un esempio potrebbe essere quello di un processo producer ad alta priorità ed uno consumer a bassa priorità. In uno scenario di buffer pieno il codice del producer (il processo in esecuzione, dato che è ad alta priorità) esegue una pausa in quanto aspetta che il consumer agisca. Il consumer tutta via non può eseguire il proprio codice in quanto , essendo a bassa priorità, non si vedrà assegnata la CPU e non andrà in esecuzione . Si crea un deadlock in quanto il producer attende una risorsa detenuta dal consumer.

Mutua esclusione :

- **Disabilitare i meccanismi di gestione delle IRQ**
- **Variabile di Lock condivisa**
- **Alternanza stretta**
- **Soluzione di Peterson**
- **Istruzione TSL(o XCHG)**