Mutua esclusione - supporto hardware

Index

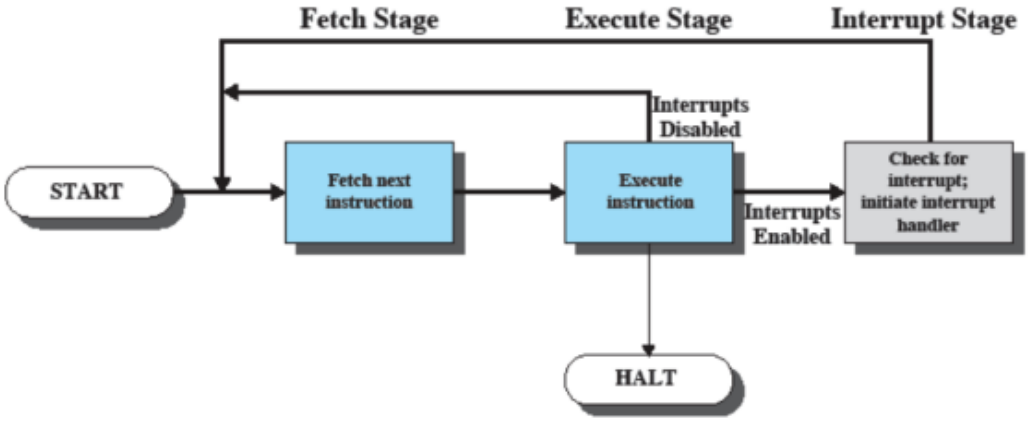
Introduction

In questa sezione vedremo dei modi funzionanti (a differenza della sezione precedente) per far rispettare la mutua esclusione

---

Disabilitazione delle interruzioni

```
while (true) {
        /* prima della sezione critica */;
        disabilita_interrupt() ;
        /* sezione critica */;
        riabilita_interrupt() ;
        /* rimanente */;
}
```



Disabilitando le interruzioni evito che il dispatcher interrompa il processo mentre si trova all'interno della sezione critica, però ci sono diversi problemi.

Uno dei problemi più evidenti è che, se questa possibilità fosse concessa a tutti i processi utente, questi ne abusino riducendo così la multiprogrammazione.
Un ulteriore problema è che questo metodo funziona localmente sul singolo processore, quindi disabilitare le interruzioni su un singolo processore, non le disabilita sugli altri; quindi un altro processo potrebbe accedere alla sezione critica semplicemente perché viene messo in esecuzione su un altro processore, eseguendo una corsa critca

---

Istruzioni macchina speciali

Per risolvere i due problemi sopracitati potrei utilizzare delle istruzioni macchina speciali come `compare_and_swap` e la `exchange` entrambe **atomiche** (l'hardware garantisce che un solo processo per volta possa eseguire una chiamata a tali funzioni/interruzioni anche se ci sono più processori)

`compare_and_swap`

Se il valore di `word` è uguale al valore di `testval` allora cambio il valore di `word` in `newval` e ritorno in ogni caso il precedente valore di `word`

```
int compare_and_swap(int word, int testval, int newval) {
        int oldval;
        oldval = word;
        if (word == testval) word = newval;
        return oldval
}
```

Mutua esclusione

```
/* program mutualexclusion */
const int n = /* number of processes */
int bolt;
void P(int i) {
        while (true) {
                while (compare_and_swap(bolt, 0, 1) == 1) /* do
nothing */
                /* critical section */
                bolt = 0;
                /* remainder */
        }
}

void main() {
        bolt = 0;
        parbegin(P(1), P(2), ..., P(n));
}
```

`compare_and_swap` prende la variabile `bolt`, vede se è 0, se vale 0 gli assegna 1
Se viene mandato in esecuzione un secondo processo questo non uscirà mai dal `while` finché il primo processo non uscirà dalla sezione critica
Potrebbe però succedere che dopo aver impostato `bolt` a 0 vada avanti e ritorni nella sezione critica, lasciando il secondo processo in attesa (starvation)

> ⚠ **Warning**
>
> E' importante che tra il controllare che `bolt` sia 0 e metterlo a 1 non ci possano essere interferenze

exchange

La funzione `exchange` ha il compito di scambiare il contenuto di due argomenti (indirizzi di memoria)

```
void exchange(int register, int memory) {
        int temp;
        temp = memory;
        memory = register;
        register = temp;
}
```

Mutua esclusione

```
/* program mutualexclusion */
const int n = /* number of processes */
int bolt;
void P(int i) {
        while (true) {
                int keyi = 1;
                do exchange(keyi, bolt)
                while (keyi != 0);
                /* critical section */
                bolt = 0;
                /* remainder */
        }
}

void main() {
        bolt = 0;
        parbegin(P(1), P(2), ..., P(n));
}
```

Il primo processo ad entrare scambierà 1 con 0 e visto che `1 != 0` uscirà dal while entrando nella sezione critica. Mentre il secondo processo continuerà a tentare di scambiare `keyi` e `bolt` ma saranno entrambi 1 finché il primo processo non uscirà dalla sezione critica impostando quindi `bolt` a 0

Vantaggi

Ci sono dei vantaggi nell'applicare queste istruzioni macchina speciali:

- sono applicabili a qualsiasi numero di processi, sia su un sistema ad un solo processore che ad un sistema a più processori con memoria condivisa
- semplici e quindi facili da verificare
- possono essere usate per gestire sezioni critiche multiple

Svantaggi

Però hanno anche degli svantaggi:

- sono basate sul *busy-waiting* (spreco di tempo di computazione), e il ciclo di busy wait non è distinguibile da codice "normale" quindi la CPU lo deve eseguire fino al timeout (oppure ci deve essere più di una CPU)
- possibile la starvation
- possibile il deadlock, se a questi meccanismi viene abbinata la priorità (fissa) se un processo A a bassa priorità viene interrotto mentre è già nella sezione

critica e un processo B a priorità alta entra nel busy waiting, B non può essere interrotto per eseguire A a causa della priorità, e A non può andare avanti perché solo B, finendo la sua sezione critica, lo può far uscire dal busy-waiting