

18-04-2023

Analisi del semaforo MUTEX

- Sono utili nella **gestione delle risorse condivise** per la mutua esclusione.
- Sono validi quando si fa uso di **thread implementati nello spazio utente**:
 - **non vengono riconosciuti dal Kernel** e quindi non sono soggetti a scheduling
 - Eventuale **scheduling è gestito dallo stesso processo**
 - Quando un **thread si blocca, blocca l'intero processo** visto che il kernel non li conosce

Mutex si può trovare in 2 stati: **locked** (valore 1) e **unlocked** (valore 0)

Si usa `mutex_lock()` e `mutex_unlock()`:

```
mutex_lock:
    TSL REGISTER,MUTEX -- Mutex viene copiata in REGISTER e viene incrementato il valore
    CMP REGISTER,#0
    JZE ok --Accede alla sezione critica se MUTEX = 0
    CALL thread_yield
    JMP mutex_lock
ok:RET

mutex_unlock:
    MOVE MUTEX,#0
    RET
```

Sono molto simili a `enter_region/leave_region` ma:

- devono essere possibili l'uso delle istruzioni TSL/XCHG visto che si è a **livello utente**
- Non si ha busy waiting se si usa MUTEX

Differenze fra le 2 versioni:

```
enter_region:
    TSL REGISTER,LOCK
    CMP REGISTER,#0
    JNE enter_region
    RET
```

```
mutex_lock:
    TSL REGISTER,MUTEX
    CMP REGISTER,#0
    JZE ok
    CALL thread_yield
    JMP mutex_lock
ok:RET
```

Cede la CPU ad un altro thread

- **Sinistra:** Se il CMP fallisce allora avviene l'iterazione infinita fino a quando viene soddisfatta la condizione
- **Destra:** I thread a livello utente non sono conosciuti dal kernel, allora non sono soggetti a scheduling dal kernel ma è fatto dal processo stesso.
 - Si ha un **vantaggio** perchè si può usare uno **scheduling personalizzato** ma il processo non ha il clock che ha il kernel e quindi non si può decidere per quanto tempo eseguire un processo. **E' il processo stesso a rilasciare la CPU volontariamente**

- Se si trova **MUTEX occupato** (locked), allora si esegue `CALL thread_yield` ed è la chiamata di sistema che **rilascia la CPU volontariamente** e **viene eseguito un altro thread dello stesso processo**.
- Questa operazione è veloce perchè **gestita dallo stesso processo**

*Quindi `mutex_lock` e `mutex_unlock` non fanno chiamate al kernel, quindi si **velocizzano le varie operazioni**. Esse sono molto utili a **LIVELLO UTENTE***

Futex

Gli spin-lock, pur consumando la CPU, può essere breve ma se non lo è si ha lo spreco dell'unità di elaborazione

Futex è una caratteristica tipica in Linux:

- un sistema è in grado di gestire numerose contese nelle parti condivise e il kernel si occupa di risvegliarlo
- diventa efficiente quando le contese sono frequenti. Se non lo fossero, le chiamate al kernel (più costose) penalizzano e aumentano il costo
- implementa un sistema lock, simile a MUTEX, senza attivare il kernel e quindi non coinvolgerlo quasi mai
- E' formato da 2 parti:
 - **servizio kernel**: che fornisce una coda di attesa che consente a più processi di andare in pausa (lock). E' il kernel stesso che li sblocca in **maniera esplicita**. Attraverso una chiamata di sistema si fanno tali operazioni.
 - **libreria utente**:

FUTEX agisce interamente nello spazio utente:

- se `lock = 1`, un thread acquisisce il lock e accede alla sezione critica
- quindi lock viene decrementato così da garantire la mutua esclusione
- Se lock è già occupato da un altro, quindi vale 0, allora la libreria FUTEX, **piuttosto che fare busy waiting**, allora questo thread **viene messo nella coda di attesa** che si trova nel kernel tramite *chiamata di sistema*
- Quando lock = 1 (*di nuovo*), allora il kernel **risveglia** il thread che è stato messo in attesa

Se non ci sono contese alla variabile lock, allora non ci sono chiamate al kernel e quindi le esecuzioni sono più veloci

Se ho contese alla variabile lock, allora probabilmente devo fare chiamate di sistema al kernel

Monitor

Soluzione efficiente per gestire la **mutua esclusione**:

- è una *raccolta di variabili*, procedure ecc che sono raggruppate in un **PACCHETTO**
- sono dei *costrutti* in un determinato linguaggio di programmazione

*Per esempio, con la **OOP** ci sono parti private (variabili, strutture dati e qualche metodo) e pubbliche (tutto il resto)*

Quando si vuole accedere a parti private di un oggetto si ottiene un errore visto che esse possono essere lette solo dai metodi pubblici nello stesso oggetto.

La stessa cosa avviene con i **Monitor**:

- Il compilatore sa che la struttura dati (l'insieme dei dati) è una parte speciale e quindi va **gestita in modo diverso rispetto ad altri metodi/procedure**
- **Tutti i processi** possono chiamare le **procedure di un monitor** ma **non si può accedere alla struttura interna** di questo mondo se non attraverso le procedure offerte dal monitor stesso
- Per ottenere una **corretta mutua esclusione**, in ogni istante **può essere attivo un solo processo alla volta** all'interno di questo mondo
- La prima istruzione, infatti, verifica che non ci sia nessun altro processo che sia attivo
- Se nessun processo sta usando il monitor, allora quel processo stesso può usare le procedure del monitor

Mutua esclusione

La mutua esclusione è **gestita dal compilatore** e non dal programmatore visto che viene gestita tramite una **MUTEX** o tramite un **semaforo binario**

Con i monitor è possibile **gestire le race conditions** convertendo le regioni critiche in **procedure di monitor** -> si ha che **MAI due processi si trovano allo stesso tempo in esecuzione all'interno del monitor**.

Nel caso produttore-consumatore come si blocca un processo che non può proseguire? (buffer pieno) Come si gestisce la sincronizzazione?

- si usano **variabili condizioni**, insieme alle operazioni **wait** e **signal**, permettono di gestire la **sincronizzazione delle operazioni**
- si deve gestire quando un processo può eseguire la sua operazione e quando non è possibile.
 - La mutua esclusione è gestita dai monitor in modo automatico tramite la **conversione delle regioni critiche in procedure del monitor stesso**
- la **wait sospende il processo chiamante** finché non viene invocata la sua alternativa, cioè la **signal che risveglia il processo addormentato da wait**
 - Se viene invocata **signal** e non ci sono processi bloccati, allora non si avranno effetti

Produttore-consumatore

Si deve gestire che il produttore può eseguire la sua operazione finché il buffer non è pieno e viceversa per il consumatore.

- Si esegue una **wait su una variabile condizione** (per esempio la **full** che si era usata come semaforo). Se è piena allora il processo chiamante viene bloccato e in questo caso si può consentire agli altri di proseguire le proprie operazioni.
- quindi il consumatore può operare che, più avanti, userà **signal sulla variabile condizione (full)**

*Cosa accade dopo signal? per evitare che **2 processi siano attivi allo stesso tempo nel monitor**, allora si deve decidere chi deve continuare la sua operazioni dopo la signal.*

1. Il **metodo Hoare**: il processo che deve essere eseguito è quello **APPENA SVEGLIATO**
2. Il **metodo Brinch-Hansen**: il processo che esegue **signal** deve **uscire subito dal monitor** e rappresenta proprio **la sua ultima istruzione** che appare all'interno di quel processo. Se viene fatto una **signal** come ultima istruzione di una procedura su cui **ci sono in attesa più processi** su questa

variabile condizione, verrà **selezionata uno di questo "in attesa"** come **nuovo processo risvegliato**

3. L'ultimo metodo: si **manda la signal** ma **in esecuzione continua a rimanere chi ha eseguito la signal**. Chi si è appena svegliato **continua ad aspettare che il chiamante termini la sua esecuzione**

La wait deve arrivare prima di una signal altrimenti verrà persa

Differenze cruciali fra sleep, wakeup

Quando la wakeup andava a vuoto, con i monitor questo non può succedere perchè la mutua esclusione è garantita dalle stesse procedure dei monitor

- `wait` -> **sospende** un processo (in caso di regione critica occupata) finchè non viene riattivato
- `signal` -> **risveglia** un processo sospeso, quindi in `wait()`. Se non c'è nessun processo in wait allora questa chiamata va a vuoto

Esempio: Dato un processo P che esegue una `signal(x)` e un processo Q che è in `sleep(x)`. Entrambi i processi possono andare in esecuzione ma chi ci va? Continua P o si esegue Q?

- **Hoare: signal & wait** -> P viene sospeso (`sleep`) e Q va in esecuzione
- **Mesa: signal & continue** -> Q riceve la signal ma aspetta che P termina la sua operazione
- **Compromesso: signal & return** -> usato con *concurrent Pascal* cioè: P fa signal, lascia il monitor a Q che viene immediatamente ripreso

Molti linguaggi di programmazione gestiscono i monitor (C#, Java) mentre altri non hanno i monitor (C) e in questo caso va implementato diversamente.

*In Java si usa la parola chiave **SYNCHRONIZED** per identificare un metodo monitor e vuol dire che nessun altro thread può eseguire un altro metodo synchronized dello stesso oggetto*

Svantaggi dei monitor

- Non sono gestiti da tutti i linguaggi di programmazione e quindi non sono gestiti dal compilatore

Esempio produttore-consumatore con i monitor

```
monitor pc_monitor
condition full, empty;
integer count = 0;

function insert(item)
  if count = N then wait(full);
  insert_item(item);
  count = count + 1;
  if count = 1 then signal(empty)

function remove()
  if count = 0 then
    wait(empty);
  remove_item(item);
  count = count - 1;
  if count = N-1 then signal(full)

function producer()
  while (true) do
    item = produce_item()
    pc_monitor.insert(item)

function consumer()
  while (true) do
    item = pc_monitor.remove()
    consume_item(item)
```

- **count** gestisce la **variabile condizione** e indica il *numero di elementi nel buffer*
- Si usano 4 metodi di una classe `pc_monitor`
- Viene usato il **modello Hoare**, cioè **signal & wait**
- Si usano le variabili condizione `full` e `empty` -> **full** serve per sospendere il produttore, **empty** serve per sospendere il consumatore

In questo esempio si è applicato il metodo Brinch-Hansen