7° lezione S.O.

Due Date	@March 28, 2025
Multi-select	Semafori
Status	Done

Istruzioni TSL

L'istruzione **Test-and-Set Lock (TSL)** permette, in un'unica operazione atomica, di testare e impostare una variabile di lock. Questa operazione è utilizzata per bloccare l'accesso a una specifica struttura dati, garantendo la sincronizzazione tra processi.

L'istruzione **TSL** ha due argomenti:

- **L'indirizzo della variabile di lock** (una variabile condivisa che, per convenzione, assume il valore `0` quando la struttura è libera).
- **Un registro della CPU**, utilizzato per memorizzare il valore precedente della variabile di lock.

Funzionamento

L'operazione eseguita da TSL è simile a un'istruzione `MOV`, ma con una differenza fondamentale:

1. Copia il valore della variabile di lock nel registro.
2. Imposta la variabile di lock a un valore prestabilito (di solito `1`).

Tuttavia, la maggior parte dei problemi legati alle sezioni critiche nasce dal fatto che viene imposta la prelazione nei momenti non opportuni. L'idea quindi è quella di rendere l'impostazione della variabile di lock atomica, un'azione che non si può scindere, e il fatto di avere un'unica istruzione dà, di base, una certa garanzia: non potrà mai essere divisa dalla prelazione.

D'altra parte, questa istruzione **TSL** (Test and Set Lock) esegue due operazioni in memoria. Di conseguenza, il loro accodamento potrebbe essere

problematico. Una sua particolarità, però, è che la CPU blocca il bus di accesso alla memoria per tutti gli altri core, impedendo interlacciamenti in modalità problematiche. Questo garantisce un'atomicità totale senza introdurre tempi morti(?). Inoltre, è un'istruzione eseguibile anche in modalità utente. La TSL, di per sé, non svolge altre operazioni oltre a queste due.

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

leave_region:
   MOVE LOCK,#0
RET
```

L'istruzione **TSL** viene inserita nella funzione `enter_region()`. Successivamente, si controlla se il valore precedente del lock è `0` o `1`. Se il valore non è `0`, viene eseguito un **jump not equal**, facendo sì che il ciclo continui fino a quando il lock non diventa disponibile. Questo approccio si basa sul **busy waiting**: quando il ciclo termina e si arriva al `return`, significa che l'accesso alla sezione critica è consentito.

La funzione `leave_region()` si limita a impostare la variabile di lock a `0`, rilasciando così la risorsa.

Questa soluzione è utilizzata dal SO stesso per garantire una mutua esclusione, la sfrutta anche il kernel perchè ha garanzie che funziona anche sui sistemi multicore. Tuttavia il SO sei si trova su un sistema mono-core, utilizza un’altra soluzione, quella di disabilitare gli interrupt.

Un istruzione simile presente su altre architetture e la struttura XCHG, Prende sempre un registro e una variabile di lock

```
MOV registro, 1
XCHG registro, lock
```

è la stessa cosa di fare TSL

L’unico problema che rimane da risolvere è il busy waiting

Uno **spinlock** è un lock implementato tramite busy waiting. Vanno oltre il semplice spreco e possono diventare sintomi di malfunzionamento.

Esempio

Si ha una priorità di esecuzione. Suppiamo di avere tre processi:

1. P_H
2. P_M
3. P_L

I tre processi lavorano su una struttura dati condivisa usando una variabile di lock tramite la TSL.

Come viene gestita la priorità?

La strategia più semplice è quella di considerare tutti i processi nella coda dei processi pronti e selezionare quello con la priorità più alta.

Finché il processo P_H è presente, gli viene assegnata la CPU. Quando decide di fare altro o si blocca, allora P_M e P_L possono utilizzare la CPU in base alla loro priorità.

l processo P_L (a priorità più bassa) verrà eseguito solo quando sia P_H che P_M sono bloccati o non hanno più nulla da eseguire. Questo avviene perché il sistema di scheduling seleziona sempre il processo con la priorità più alta tra quelli pronti.

Quindi, se P_H è pronto, avrà sempre la CPU. Se PH si blocca o termina, allora verrà eseguito P_M. Solo quando anche P_M è bloccato o terminato, allora P_L potrà essere eseguito.

Si ipotizza anche che non vienga utilizzata la prelazione. Si suppone che P_L sia il 1° processo ad accedere alla varaibile lock e in particolare entra dentrio questa. Ciò vol dire che si immagine che questi due processi siano bloccati. P_L ha lo possibilta di utilizzare la struttura dati condivisa, in un certo momento P_H o anche P_M si risvegliano e P_H acquisirà la CPU e comincia a controllare la variabile di lock fin quando entra nella sezione critica.

A questo punto, P_H sta continuamente controllando la variabile di lock mentre P_L sta ancora eseguendo la sua sezione critica. Dato che non è presente la prelazione, P_L continuerà a occupare la CPU finché non rilascerà la variabile di lock.

Quando P_L termina la sua sezione critica e rilascia la lock, P_H, che sta attivamente controllando la variabile, riesce immediatamente ad acquisirla e a entrare nella sezione critica. Questo accade perché P_H ha la priorità più alta e, una volta che la lock è disponibile, prende immediatamente il controllo della risorsa condivisa.

Il problema principale di questo comportamento è che, mentre P_H stava aspettando, ha consumato inutilmente cicli di CPU verificando costantemente la lock, anziché lasciare che la CPU fosse utilizzata in modo più efficiente da altri processi. Questo è un effetto indesiderato dello **spin lock**, in cui un processo con priorità più alta spreca risorse controllando ripetutamente una variabile invece di attendere in modo passivo.

Se invece fosse stato adottato un meccanismo di sincronizzazione basato su **attesa bloccante** (come un mutex con attesa gestita dal sistema operativo), P_H sarebbe rimasto bloccato senza occupare inutilmente la CPU, migliorando l'efficienza complessiva del sistema.

Stato Passivo del Processo

Quando un processo va in pausa in modo passivo, significa che transita in uno stato particolare in cui il sistema operativo si occupa di non assegnargli la CPU finché rimane bloccato. Questa operazione può essere gestita esclusivamente dal sistema operativo, motivo per cui il problema si può risolvere chiedendo il suo intervento. È quindi necessario introdurre delle primitive di sistema offerte dal sistema operativo, appositamente progettate per gestire queste situazioni in modo efficiente.

- `wake_up()`: con indicazione di quale sarà il processo o thread da svegliare e lo farà
- `sleep()` : fa addormentare il processo chiamante

Il Problema Produttore-Consumatore

Il problema del produttore-consumatore è un classico scenario della programmazione concorrente. Esso coinvolge due tipi di processi: i produttori, che generano dati e li inseriscono in un buffer condiviso, e i consumatori, che prelevano dati dal buffer per elaborarli.

Il problema si verifica quando il buffer è pieno e il produttore deve attendere, oppure quando il buffer è vuoto e il consumatore non ha nulla da prelevare.

Il produttore crea un elemento, il cui contenuto specifico non è rilevante per la logica del problema, e il consumatore è interessato a prelevarlo. Tuttavia, il passaggio dell'elemento non avviene in modo diretto, ovvero il produttore non lo porge immediatamente al consumatore attendendo che quest'ultimo lo prenda. Questo eviterebbe il rischio di dover gestire una sincronizzazione perfetta, che renderebbe il sistema meno efficiente. Per questo motivo, viene utilizzato un buffer intermedio limitato che permette di disaccoppiare la produzione dal consumo.

Produzione

Il produttore genera un nuovo elemento e verifica se c'è spazio disponibile nel buffer. Se il buffer è pieno, il produttore si mette in attesa finché non si libera spazio. Una volta possibile, inserisce l'elemento nel buffer e notifica il consumatore, che può riprendere l'elaborazione dei dati.

Consumo

Il consumatore verifica se ci sono elementi nel buffer. Se il buffer è vuoto, si mette in attesa finché un produttore non aggiunge un nuovo elemento. Quando un elemento è disponibile, lo preleva e lo elabora. Dopo il consumo, il consumatore notifica il produttore, che può così inserire nuovi elementi nel buffer senza superare la capacità disponibile.

Estensione a Più Produttori e Consumatori

Il modello può essere ampliato con più produttori e consumatori, aumentando la complessità della sincronizzazione. In questo scenario, è fondamentale gestire correttamente l'accesso concorrente al buffer per evitare sovraccarichi o blocchi del sistema. Inoltre, è necessario monitorare il numero di elementi presenti nel buffer e garantire una corretta gestione delle priorità tra i processi coinvolti.

```
function producer()
    while (true) do
        item = produce_item()  // Produce un nuovo elemento

        if (count == N) then   // Se il buffer è pieno, il produttore si addormenta
            sleep()
```

```
        insert_item(item)      // Inserisce l'elemento nel buffer
        count = count + 1      // Incrementa il numero di elementi nel buffer

        if (count == 1) then   // Se il buffer era vuoto e ora contiene un elemento,
            wakeup(consumer)


function consumer()
    while (true) do
        if (count == 0) then   // Se il buffer è vuoto, il consumatore si addormenta
            sleep()

        item = remove_item()   // Preleva un elemento dal buffer
        count = count - 1      // Decrementa il numero di elementi nel buffer

        if (count == N - 1) then  // Se il buffer era pieno e ora c'è spazio libero, sv
            wakeup(producer)

        consume_item(item)     // Consuma l'elemento prelevato
```

Applicazioni Reali

Il problema produttore-consumatore trova applicazione in diversi ambiti. È utilizzato nella gestione della memoria nei sistemi operativi, nella comunicazione tra thread nei programmi concorrenti e nell'elaborazione dei dati in sistemi distribuiti. L'uso di tecniche di sincronizzazione consente di gestire in modo efficiente le risorse condivise, evitando sprechi e garantendo un funzionamento stabile del sistema.

Questa soluzione assicura che produttori e consumatori operino in modo coordinato, prevenendo situazioni di blocco e ottimizzando l’uso delle risorse disponibili.

Esempio del Problema Produttore-Consumatore con Meccanismo di Sveglia

Consideriamo lo scenario classico del problema **Produttore-Consumatore**, in cui un **consumatore** può trovarsi nella condizione di addormentarsi (cioè entrare in attesa) in assenza di elementi nel buffer. Il risveglio di tale

consumatore è governato da un evento (es. una notifica da parte del produttore) che segnala la disponibilità di un nuovo elemento.

Problema dell'Evento Prematuro

Supponiamo ora che il consumatore stia per addormentarsi, ovvero stia entrando in uno stato di attesa bloccante. Cosa accade se, **proprio in quel momento**, arriva un evento (una "wakeup")?

In questo caso, il consumatore riceve la notifica di risveglio, ma non è ancora del tutto addormentato. L'effetto pratico è che l'evento viene **perso**: il consumatore non è ancora in attesa, quindi non "aggancia" l'evento, e quando successivamente entra nello stato dormiente, **non riceverà alcun risveglio**, perché l'evento è già stato emesso. In sostanza, il risveglio è avvenuto in modo prematuro e non è stato correttamente registrato.

Problema dell'Evento Ritardato

Viceversa, se il consumatore **si addormenta completamente** prima che il produttore inserisca un elemento nel buffer, rischia di **non accorgersi** dell'inserimento. Rimarrà dormiente fino a che non si verifichi un'altra notifica o meccanismo di risveglio, potenzialmente ritardando il consumo dell'elemento.

In casi particolari, anche il produttore potrebbe trovarsi in uno stato simile: se il buffer è pieno e il produttore sta per addormentarsi, ma nel frattempo il consumatore rimuove un elemento, il produttore potrebbe **non percepire l'evento**che lo avrebbe dovuto risvegliare, generando una condizione analoga ma speculare.

Problema Fondamentale

Tutti questi casi evidenziano un **problema di sincronizzazione** legato alla gestione degli eventi e degli stati di sonno/veglia dei processi concorrenti. Il punto cruciale è che un evento (es. una "wakeup") potrebbe essere:

- **Perso** se arriva troppo presto (prima che il processo sia effettivamente dormiente);
- **Ignorato** se il processo non controlla correttamente il proprio stato al momento della ricezione.

Una possibile, ma poco elegante, soluzione sarebbe quella di implementare un controllo ciclico (polling) ad ogni ciclo di I/O, per verificare la disponibilità di

eventi. Tuttavia, questo approccio risulta inefficiente e non soddisfacente.

Soluzione: Flag di Wakeup Persistente

Una soluzione più robusta consiste nell'introdurre un **meccanismo di sveglia persistente**, mediante un flag condiviso:

- Se una "wakeup" viene emessa mentre il processo **non è ancora dormiente**, essa non viene persa, ma viene **registrata** mediante l'attivazione di un flag.
- Al momento dell'effettiva "sleep", il processo controllerà questo flag. Se risulta attivo, il processo **eviterà di dormire**, oppure **si risveglierà immediatamente**.
- Il processo ha quindi la responsabilità di "consumare" questo flag, anche se l'evento è arrivato troppo presto, garantendo così la coerenza dello stato.

Fogli di Bloccaggio (Locking Sheets) e Semafori

I **fogli di bloccaggio**, ovvero i **semafori**, operano come garanti del rispetto di una determinata condizione: una **invariante booleana** che non può essere violata. Tale condizione rappresenta una regola che tutte le operazioni coinvolte devono osservare. In questo senso, il semaforo rappresenta un meccanismo di controllo fondamentale nei contesti concorrenti.

Meccanismo di Funzionamento: `wait` e `signal`

I semafori si fondano su due operazioni principali:

- `wait` (o `P`, o `down`): decrementa il valore del semaforo;
- `signal` (o `V`, o `up`): incrementa il valore del semaforo.

L'operazione `signal`, per essere valida, presuppone che vi sia almeno un processo in attesa: essa **risveglia un processo**, se presente, altrimenti incrementa semplicemente il contatore. Tale incremento non è arbitrario, ma condizionato al significato del semaforo: se è attivo almeno un processo in attesa, l'incremento ha un effetto immediato sullo stato del sistema.

Semaforo come Astratto Meccanismo di Sveglia

Questa logica è strettamente legata al concetto precedentemente trattato di **evento di risveglio**. Il semaforo, infatti, può essere visto come un'**astrazione generalizzata della sveglia**: la sua struttura interna consente di conservare

l'informazione che un evento è accaduto, anche se non immediatamente percepito dal processo.

Atomicità delle Operazioni su Semafori

Un semaforo è, dal punto di vista implementativo, una **variabile intera non negativa**, soggetta a due operazioni **atomiche**: `down` e `up`. L'atomicità è fondamentale per garantire il corretto comportamento in ambienti concorrenti. L'operazione `down`, se non può essere completata (cioè se il semaforo è a zero), mette il processo in pausa passiva (il processo viene sospeso senza consumare CPU).

Questo scenario evidenzia come l'**evento software** (come la sveglia) sia strettamente legato al comportamento dei processi. Se più processi accedono alla stessa variabile intera **senza semafori**, possono nascere condizioni di race: ad esempio, due processi leggono il valore 1 e decidono entrambi di procedere con `down`, producendo un'incoerenza.

Necessità di Accesso Concorrente Sicuro

Questo implica la necessità di un **accesso controllato e garantito** alle primitive. Se un processo esegue una `down`, e un altro contemporaneamente fa lo stesso, il sistema deve assicurare che **una sola operazione venga completata**, mentre l'altra viene sospesa. Solo una `up` successiva potrà consentire l'avanzamento del processo sospeso.

Non è sufficiente che la `up` venga completata prima o dopo; è necessario che l'intera sequenza sia **atomica**. Se l'operazione viene interrotta a metà, e ripresa da un altro processo su un altro semaforo, il sistema non può più garantire coerenza.

Implementazioni Atomiche e Disabilitazione degli Interrupt

La garanzia di atomicità può essere assicurata in vari modi. Una strategia consiste nella **disabilitazione temporanea degli interrupt** durante l'esecuzione delle operazioni critiche sul semaforo. Questa tecnica è efficace nei sistemi **single-core**, poiché evita che il controllo passi ad altri processi in momenti cruciali.

Nei sistemi **multi-core**, tuttavia, la disabilitazione degli interrupt non è sufficiente. È necessario l'uso di **meccanismi di locking avanzati**, come **spinlock** o istruzioni atomiche come `fetch-and-add`, che permettono di garantire un accesso esclusivo anche tra più core.

Implementazione di Semafori nel Kernel e Gestione della Mutua Esclusione

L'esecuzione delle operazioni `wait` e `signal` nel kernel necessita di un meccanismo atomico. Tipicamente, nel kernel si utilizza la **disabilitazione temporanea degli interrupt** per evitare che il flusso di esecuzione venga interrotto. Questo garantisce che la verifica del valore del semaforo, il suo aggiornamento e l'eventuale gestione della coda dei processi in attesa avvengano senza interruzioni.

Nel contesto di sistemi multi-core, la mutua esclusione richiede meccanismi più sofisticati per garantire che l'accesso alla struttura dati del semaforo non avvenga in modo concorrente da più processori.

- L'operazione `wait` decrementa il valore del semaforo. Se il risultato è negativo, il processo viene **bloccato** e inserito in una **coda di attesa**.
- L'operazione `signal` incrementa il valore del semaforo. Se il valore è non negativo e ci sono processi in attesa, **uno viene risvegliato**.

Questo modello consente un blocco **passivo**, in cui i processi sospesi **non consumano risorse di CPU**.

Mutua Esclusione con Semaforo `mutex`

Per realizzare la **mutua esclusione**, si inizializza il semaforo a **1** (convenzionalmente chiamato `mutex`). Il processo che entra nella sezione critica esegue `wait(mutex)`:

- Se `mutex == 1`, l'accesso è consentito e `mutex` diventa 0.
- Se `mutex == 0`, il processo si blocca in attesa.

Al termine della sezione critica, il processo esegue `signal(mutex)`, ripristinando il valore a 1 e permettendo ad un altro processo di entrare nella sezione critica.

Semafori Generici e Sincronizzazione Produttore-Consumatore

I semafori trovano applicazione anche in scenari di **sincronizzazione più complessi**, come il **problema del produttore-consumatore**. In questo caso, si usano tre semafori distinti:

- `mutex` **(inizializzato a 1):** Garantisce l'accesso esclusivo al buffer condiviso, proteggendo le operazioni `put` e `get`.

- `empty` **(inizializzato alla dimensione del buffer):** Conta gli **slot vuoti** disponibili. Il produttore esegue `wait(empty)` per assicurarsi che ci sia spazio prima di inserire un elemento.
- `full` **(inizializzato a 0):** Conta gli **slot pieni**. Il consumatore esegue `wait(full)` per verificare la presenza di almeno un elemento prima di estrarre.

Logica del Produttore

1. Produce un elemento.
2. Esegue `wait(empty)` per controllare la disponibilità di spazio.
3. Esegue `wait(mutex)` per accedere al buffer in modo esclusivo.
4. Inserisce l'elemento nel buffer.
5. Esegue `signal(mutex)` per rilasciare il lock.
6. Esegue `signal(full)` per notificare che un nuovo elemento è disponibile.

Logica del Consumatore

1. Esegue `wait(full)` per attendere la disponibilità di un elemento.
2. Esegue `wait(mutex)` per accedere in modo esclusivo al buffer.
3. Estrae l'elemento dal buffer.
4. Esegue `signal(mutex)` per rilasciare il lock.
5. Esegue `signal(empty)` per notificare che un nuovo slot è disponibile.

Questa configurazione assicura che:

- I produttori **non inseriscano** elementi in un buffer pieno.
- I consumatori **non rimuovano** elementi da un buffer vuoto.
- L'accesso al buffer sia protetto da accessi concorrenti.

In tal modo, viene preservata l'**integrità del buffer** e garantita la corretta **sincronizzazione** delle operazioni tra produttori e consumatori.